

CMPT 2361

Computer Graphics

Shahriar Khosravi, Ph.D.

Lecture 1

Plan for this Lecture

- Introduction
- The PPM Image Format
- Our Course Policies
- Painting and Drawing
- Elements of 3D Graphics
- Hardware and Software

Shahriar Khosravi (He/Him)

- **Bachelor of Science** in Aerospace Science and Engineering (**University of Michigan**, 2012)
- **Doctor of Philosophy** in Aerospace Science and Engineering (**University of Toronto**, 2016)
 - Thesis topic: Aircraft wing design using numerical optimization based on computational fluid dynamics and finite-element structural analysis
- **Credit Risk Analytics Manager** at BMO Bank of Montreal (Toronto)
 - Data science, software engineering, and numerical modelling using the SAS and SQL scripting languages
- **Structural Dynamics and Vibrations Analyst** at Pratt and Whitney Canada (Toronto)
 - Aircraft engine design from a vibrations perspective and data science for engine component performance prediction
- **Professor of Electronics and Mechanical Engineering Technology** at Seneca College (Toronto)
 - Taught courses on the C, C++, C# programming languages, and the .NET Framework

What You Have Said in the Welcome Survey (Adjectives)

I play games somewhat competitively.

Aspiring game designer with one released game.

I once learned how to do backflips in 2 days.

Like watching the NBA.

What You Have Said in the Welcome Survey (Concerns)

No concern!

No concerns, your courses are great.

The percentage of Midterm and Final Exams.

Early morning wakeup.

Provide more practice Exams, so I can get good grades.

Heavy course load.



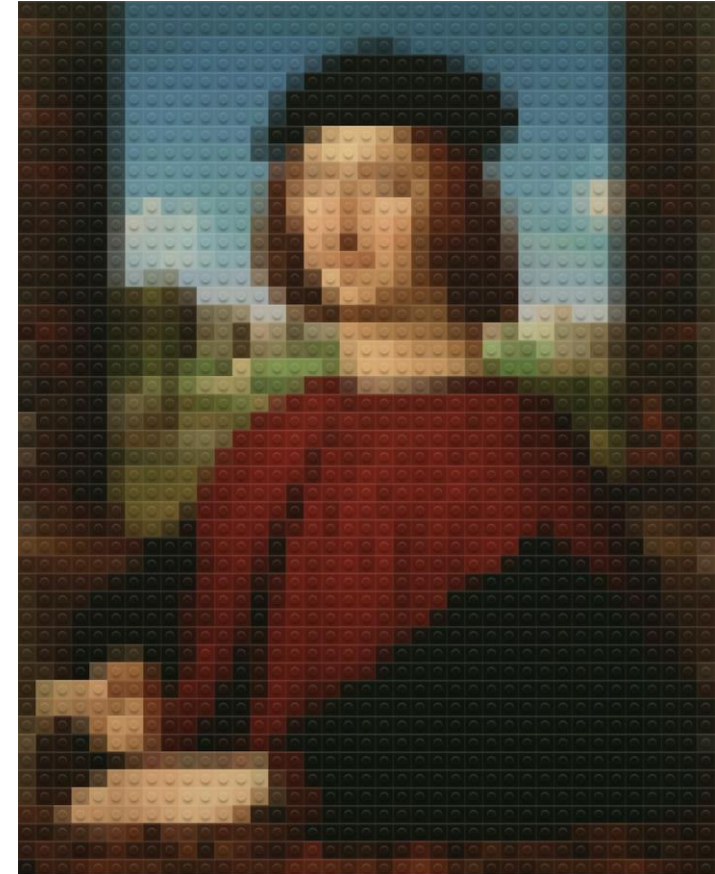
Our Course Policies



Painting and Drawing

Painting and Drawing in 2D – Pixels

- The main focus of this course is three-dimensional (3D) graphics, where most of the work goes into producing a 3D model of a scene.
- In almost all cases, the end result of a computer graphics project is a two-dimensional image.
- The direct production and manipulation of 2D images is an important topic in its own right.
- Many ideas carry over from two dimensions to three.
- Therefore, it makes sense to start with graphics in 2D.
- An image that is presented on the computer screen is made up of **pixels**.
- The screen consists of a rectangular grid of pixels, arranged in rows and columns.
- The pixels are small enough that they are not easy to see individually.



Color

- At a given time, each pixel can show only one **color**.
- Most modern screens use 24-bit color, where a color can be specified by three 8-bit numbers, giving the levels of red, green, and blue in the color.
- Any color that can be shown on the screen is made up of some combination of these three "primary" colors.
- Other formats are also possible, such as grayscale, where each pixel is some shade of gray and the pixel color is given by one number that specifies the level of gray on a black-to-white scale.
- Usually, 256 shades of gray are used.
- Early computer screens used indexed color, where only a small set of colors, usually 16 or 256, could be displayed.
- For an indexed color display, there is a numbered list of possible colors, and the color of a pixel is specified by an integer giving the position of the color in the list.

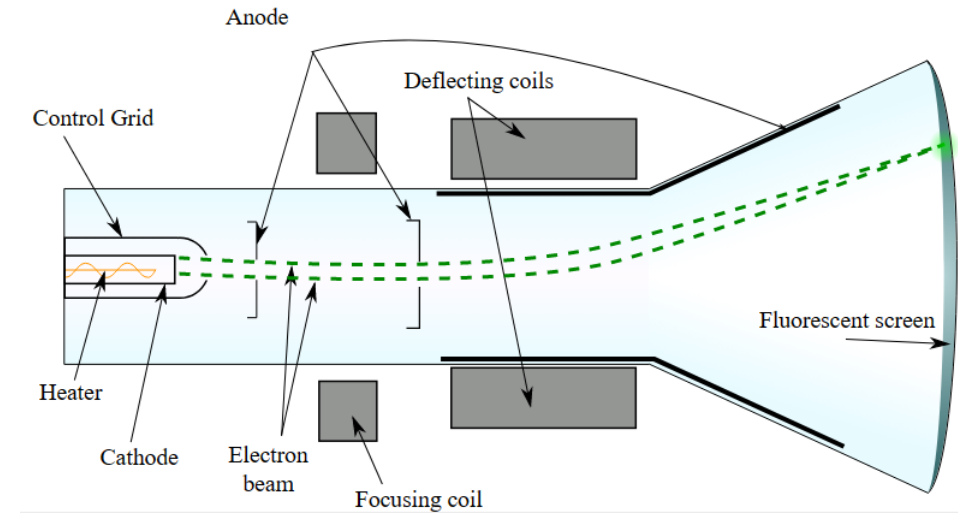


Frame Buffer

- The color values for all the pixels on the screen are stored in a large block of memory known as a **frame buffer**.
- Changing the image on the screen requires changing color values that are stored in the frame buffer.
- The screen is redrawn many times per second, so that almost immediately after the color values are changed in the frame buffer, the colors of the pixels on the screen will be changed to match, and the displayed image will change.
- A computer screen used in this way is the basic model of **raster graphics**.

Raster Graphics

- The term "**raster**" technically refers to the mechanism used on older vacuum tube computer monitors:
 - An electron beam would move along the rows of pixels, making them glow.
 - The beam was moved across the screen by powerful magnets that would deflect the path of the electrons.
 - The stronger the beam, the brighter the glow of the pixel, so the brightness of the pixels could be controlled by modulating the intensity of the electron beam.
 - The color values stored in the frame buffer were used to determine the intensity of the electron beam.
 - For a color screen, each pixel had a red dot, a green dot, and a blue dot, which were separately illuminated by the beam.



Raster Graphics

- A modern flat-screen computer monitor is not a raster in the same sense.
- There is no moving electron beam.
- The mechanism that controls the colors of the pixels is different for different types of screen.
- The screen is still made up of pixels, and the color values for all the pixels are still stored in a frame buffer.
- The idea of an image consisting of a grid of pixels, with numerical color values for each pixel, defines raster graphics.
- LCD (liquid-crystal display) televisions have millions of tiny picture elements called pixels that can be switched on or off electronically to make a picture.
- Each pixel is made up of three smaller red, green, and blue sub-pixels. These can be individually turned on and off by liquid crystals.

The Portable Pixel Map (PPM) Image Format

The PPM Image Format

- Portable Pix Map (PPM) is a color image format, designed specifically to be extremely easy to learn and write programs for.
- A PPM image represents a Red-Green-Blue (RGB) colored picture.
- Each pixel's color is represented by three consecutive integers for Red-Green-Blue (RGB).
- You could use [GIMP](#) to view the image.



```
P3
# example PPM image
4 4
7
7 0 0    0 7 0    0 0 7    7 7 7
7 7 0    7 0 7    0 7 7    0 0 0
1 0 5    5 5 1    7 1 4    1 5 6
5 5 1    6 4 7    1 1 4    2 2 7
```

The PPM Image Format

- Portable Pix Map (PPM) is a color image format, designed specifically to be extremely easy to learn and write programs for.
- A PPM image represents a Red-Green-Blue (RGB) colored picture.
- Each pixel's color is represented by three consecutive integers for Red-Green-Blue (RGB).
- You could use [GIMP](#) to view the image.



```

P3
# example PPM image
4 4
7
7 0 0 0 7 0 0 0 7 7 7
7 7 0 7 0 7 0 7 7 0 0 0
1 0 5 5 5 1 7 1 4 1 5 6
5 5 1 6 4 7 1 1 4 2 2 7
  
```

"Magic" Number
 Comment
 Width and Height
 Max Color Value

Vector Graphics

- Although images on the computer screen are represented using pixels, specifying individual pixel colors is not always the best way to create an image.
- Another way is to specify the basic geometric objects that it contains, shapes such as lines, circles, triangles, and rectangles.
- This is the idea that defines **vector graphics**:
 - Represent an image as a list of the geometric shapes that it contains.
- The shapes can have attributes, such as the thickness of a line or the color that fills a rectangle.
- Not every image can be composed from simple geometric shapes.
- This approach wouldn't work for a picture of a beautiful scene with many tiny details.
- However, it works well for many types of images, such as architectural blueprints and scientific illustrations.

Vector Graphics

- Early in the history of computing, vector graphics was even used directly on computer screens.
- When the first graphical computer displays were developed, raster displays were too slow and expensive to be practical.
- It was possible to use vacuum tube technology in another way:
 - The electron beam could be made to directly draw a line on the screen, simply by sweeping the beam along that line.
- A vector graphics display would store a display list of lines that should appear on the screen. Since a point on the screen would glow only very briefly after being illuminated by the electron beam, the graphics display would go through the display list over and over to continually redraw all the lines on the list.
- To change the image, it would only be necessary to change the contents of the display list. If the display list became too long, the image would start to flicker because a line would have a chance to visibly fade before its next turn to be redrawn.

In-Class Discussion

For an image that can be specified as a reasonably small number of geometric shapes, which type of graphics should we use?

Vector or Raster?



Vector Graphics

- For an image that can be specified as a reasonably small number of geometric shapes, the amount of information needed to represent the image is much smaller using a vector representation than using a raster representation.
- Consider an image made up of one thousand line segments. For a vector representation of the image, you only need to store the coordinates of two thousand points, the endpoints of the lines. This would take up only a few kilobytes of memory.
- To store the image in a frame buffer for a raster display would require much more memory.
- Similarly, a vector display could draw the lines on the screen more quickly than a raster display could copy the same image from the frame buffer to the screen.
- **As soon as raster displays became fast and inexpensive, however, they quickly displaced vector displays because of their ability to display all types of images reasonably well.**

Painting vs. Drawing Applications

- Raster graphics and vector graphics are two different methods for creating and displaying images on a computer.
- **Painting programs** create an image by assigning colors to individual pixels, and the resulting image is represented as a grid of pixels.
 - E.g., Microsoft Paint
- **Drawing programs** create an image by adding geometric shapes, and the image is represented as a list of those shapes.
- Practical image creation and editing programs may combine elements of painting and drawing, but one or the other is usually dominant.
- Adobe Photoshop is an example of a painting program, while Adobe Illustrator is a drawing program.
- Gimp and Inkscape are free alternatives to Photoshop and Illustrator, respectively.

Graphics File Formats

- File Formats represent an image as data stored in a file.
- Some common graphics file formats for the web include **GIF**, **PNG**, **JPEG**, **WebP**, and **SVG**.
- **GIF**, **PNG**, **JPEG**, and **WebP** are raster graphics formats, which store a color value for each pixel
- **GIF** uses indexed color with a maximum of 256 colors, while **PNG** and **JPEG** can use full 24-bit color.
- Data used to represent a raster image can be large and contain redundancy, so it can be compressed to reduce its size.
- **GIF** and **PNG** use lossless compression, which means the original image can be recovered exactly from the compressed data.
- **JPEG** uses lossy compression, which means the recovered image may not be exactly the same as the original, but the difference may not be noticeable and the compressed data is usually smaller.
- **SVG** is a vector graphics format, which is an XML-based language for describing two-dimensional vector graphics images.
- Vector graphics are scalable, so there is no loss of quality when the size of the image is increased.
- **SVG** is a good choice for web browsers and **for** graphical elements on the computer's desktop, and some desktop environments now use SVG images for their icons.

Coordinate Systems

- A digital image is specified using a **coordinate system** that assigns numbers to geometric points.
- In two dimensions, each point is given a pair of coordinates, often called the x-coordinate and y-coordinate.
- A raster image is a two-dimensional grid of pixels and has a natural coordinate system using integers to specify the row and column of each pixel.
- A vector image typically uses real-number coordinates and the coordinate system used to specify the image is somewhat arbitrary.
- Coordinate systems are important in graphics and will be a major focus in this course, especially in three-dimensional graphics.



DOUGLAS COLLEGE

Elements of 3D Graphics

Geometric Modelling

- 3D graphics commonly use **geometric modeling**, where the content of an image is specified as a list of geometric objects.
- Geometric modeling involves creating an "artificial 3D world" with simple shapes arranged in 3D space and objects with attributes that determine their appearance.
- Basic shapes, or **geometric primitives**, may include points, line segments, and triangles, and complex shapes can be constructed or approximated with these basic shapes.
- To create a 2D image, the 3D scene is projected down to two dimensions, similar to taking a photograph.
- To do this, we need to figure out how to specify the following items in the scene:
 - The geometry
 - The appearance
 - The image of the scene

The Geometry

- To create a 3D scene, we start with an empty "world" with a coordinate system (x, y, z coordinates) called **world coordinates**.
- We build the scene with geometric objects specified by giving the coordinates of their basic shapes, such as the endpoints of a line segment or the vertices of a triangle.
- These basic shapes are called **geometric primitives**, and different graphics systems may have different sets available, often including only basic shapes like lines and triangles.
- To create a complex scene, we can group geometric primitives into reusable components called **geometric models** using a process called **hierarchical modeling**.
- Once a geometric model is designed, it can be used as a component in more complex models.

The Geometry – Geometric Transforms

- **Geometric transforms** adjust the size, orientation, and position of a geometric object.
- When using a model of a wheel in a model of an automobile, we can apply a single adjustment to the entire wheel model instead of adjusting the coordinates of each individual primitive.
- We can use different transforms to add multiple copies of the same model, such as adding four copies of the same wheel model to an automobile model or adding several copies of the same automobile model to a scene.
- The three most basic types of geometric transforms are **scaling**, **rotation**, and **translation**.
- Scaling adjusts the size of an object, rotation adjusts the orientation, and translation adjusts the position.
- These transforms are important in 3D graphics and can be understood more easily in 2D.

The Appearance

- To make geometric shapes interesting in 3D graphics, we assign attributes to them such as **material properties**, which determine how a surface appears and interacts with light.
- Material properties may include a basic color and other properties like shininess, roughness, and transparency.
- **Textures** are a type of material property that allow for variation in material properties from point to point on a surface.
- They are often used to add different colors to a surface by applying a 2D image as a texture, but can also refer to changes in values like transparency or "bumpiness."
- Textures can be used to add detail to a scene using fewer geometric primitives.
- The appearance of an object in 3D graphics depends on both its material properties and the **lighting** in the environment.
- To create a realistic appearance, **simulated lighting** must be added to the scene, which can have multiple sources with their own color, intensity, and direction or position.
- The interaction between the light and the material properties of the objects in the scene is important for creating a realistic appearance, and the support for lighting in a graphics system can vary in complexity.

The Image

- The ultimate goal of 3D graphics is to create 2D images of a 3D world through the process of **viewing** and **projection**.
- This involves specifying the position of the viewer, the direction the viewer is looking, and an "up" direction, and then projecting the view into 2D, similar to taking a photograph with a camera.
- The final step in 3D graphics is to assign colors to individual pixels in the 2D image. This process is called **rasterization**, and the whole process of producing an image is referred to as **rendering the scene**.
- In 3D graphics, the ultimate goal is often to create an animation, a sequence of images showing the world at different times.
- Animations can involve small changes to almost any aspect of the scene, including coordinates of primitives, transformations, material properties, and the view.
- These changes can be computed using techniques such as **physics engines**, which simulate the motion and interaction of objects based on physical laws.



DOUGLAS COLLEGE

Hardware and Software

OpenGL History

- We will be using **OpenGL** as the primary basis for 3D graphics programming.
- It was first released in 1992 by Silicon Graphics, a company known for its graphics workstations.
- Today, OpenGL is supported by the graphics hardware on most modern devices, including desktops, laptops, and mobile devices.
- Graphics processing is done by a specialized component called a **Graphics Processing Unit (GPU)**.
- GPU includes parallel processors for graphics computations and **its own dedicated memory**.
- GPU processors have faster access to data in GPU memory than CPU's access to data in main memory (RAM).
- Modern computers use GPU to improve graphics performance.
- First desktop computers managed screen contents with CPU, leading to slow graphics performance.

OpenGL API

- The GPU is responsible for carrying out graphical operations, such as drawing lines, in modern computers.
- It includes processors specifically designed for graphics computation and its own dedicated memory for storing data like images and coordinates.
- The CPU sends commands and necessary data to the GPU, which is optimized for efficiently carrying out these commands.
- OpenGL is a graphics **Application Programming Interface (API)** that most GPUs support, meaning they can understand OpenGL commands or they can be translated into commands that the GPU can understand.
- OpenGL is an open API, meaning that it is not proprietary and can be used on multiple operating systems such as MacOS and Windows.
- Other graphics APIs include **Metal** (used by Apple's MacOS) and **Direct3D** (used by Microsoft Windows), and the newer, more complex and efficient API, **Vulkan**.
- OpenGL is chosen for use in this course because it provides an easier introduction to 3D graphics and is the basis for 3D graphics in web browsers.

OpenGL API

- OpenGL was designed as a "**client/server**" system.
- The server, which is responsible for controlling the computer's display and performing graphics computations, carries out commands issued by the client.
- Typically, the server is a GPU, including its graphics processors and memory. The server executes OpenGL commands.
- The client is the CPU in the same computer, along with the application program that it is running. OpenGL commands come from the program that is running on the CPU.
- However, it is possible to run OpenGL programs remotely over a network.
- We can execute an application program on a remote computer (the OpenGL client), while the graphics computations and display are done on the computer that we are actually using (the OpenGL server).
- The key idea is that the client and the server are separate components, and there is a communication channel between those components.

OpenGL API

- The client (CPU) sends OpenGL commands and data to the server (GPU) over a communication channel.
- The capacity of this channel can impact the performance of graphics processing, as the time it takes to send data from the CPU to the GPU can limit the speed at which the GPU can process and display the image.
- Therefore, one of the driving factors in the evolution of OpenGL has been the desire to limit the amount of communication that is needed between the CPU and the GPU.
- One approach is to store information in the GPU's memory.
- If some data is going to be used several times, it can be transmitted to the GPU once and stored in memory there, where it will be immediately accessible to the GPU.
- Another approach is to try to decrease the number of OpenGL commands that must be transmitted to the GPU to draw a given image.

OpenGL API

- OpenGL draws primitives such as triangles by specifying coordinates and attributes for each vertex.
- In the original OpenGL 1.0, separate commands were required to specify each vertex and to change attribute values.
- Drawing a complex object made up of thousands of triangles would require thousands of commands.
- In OpenGL 1.1, data for an object could be loaded into arrays and transmitted to the GPU in a single step, but the data had to be retransmitted each time the object was drawn.
- **Vertex Buffer Objects** (VBOs) were introduced in OpenGL 1.5 to store coordinates or attribute values for a set of vertices in a block of memory in the GPU, allowing the data to be reused without having to be retransmitted from the CPU each time it is used.
- Similarly, OpenGL 1.1 introduced texture objects to make it possible to store several images on the GPU for use as textures. Texture images that are going to be reused several times can be loaded once into the GPU only once, and redrawn as necessary.

OpenGL, Open GL ES, and WebGL

- The OpenGL API grew over time as new capabilities were added, but the demand for new features and the complexity of the API continued to be a problem.
- OpenGL 2.0 introduced programmable graphics processing, allowing programmers to write programs (called **shaders**) to be executed on the GPU to implement new techniques without changing the API.
- The API only needed to support the ability to send these programs to the GPU for execution.
- Shaders include **vertex shaders**, which handle per-vertex computations, and **fragment shaders** (also called **pixel shaders**), which handle per-pixel computations.
- The idea of programmable graphics hardware was very successful – so successful that in OpenGL 3.0, the usual per-vertex and per-fragment processing was deprecated.
- On the embedded system side, with **OpenGL ES** 2.0 and later, the use of shaders is mandatory, and a large part of the OpenGL 1.1 API has been completely removed.
- **WebGL**, the version of OpenGL for use in web browsers, is based on OpenGL ES, and it also requires shaders to get anything at all done.

OpenGL and GLSL

- OpenGL shaders are written in **GLSL (OpenGL Shading Language)**.
- Like OpenGL itself, GLSL has gone through several versions.
- We will spend some time later in the course studying GLSL ES, the version used with WebGL and OpenGL ES. GLSL uses a syntax similar to the C programming language.
- Note that the computations that are done for different vertices are independent, and be done in parallel. The same is true of the computations for different fragments.
- GPUs can have hundreds or thousands of processors that can operate in parallel.
- The individual processors are much less powerful than a CPU, then typical per-vertex and per-fragment computations are not very complicated.
- The large number of processors, and the large amount of parallelism that is possible in graphics computations, makes for impressive graphics performance even on fairly inexpensive GPUs.



Thank you.
DOUGLASCOLLEGE

DOUGLASCOLLEGE

CMPT 2361

Computer Graphics

Shahriar Khosravi, Ph.D.

Lecture 2

Plan for this Lecture

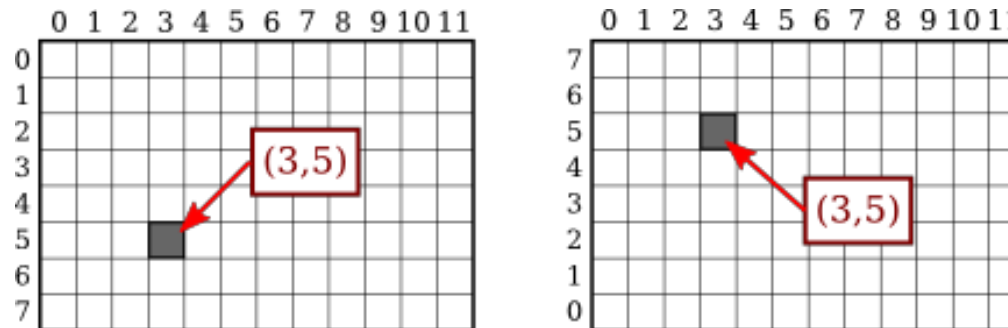
- Pixels, Coordinates and Colors
 - Pixel Coordinates
 - Real-Number Coordinate Systems
 - Aspect Ratio
 - Color Models
- Shapes
 - Basic Shapes
 - Stroke and Fill
 - Polygons, Curves, and Paths
- Transforms
 - Viewing and Modeling
 - Translation
 - Rotation
 - Combining Transformations
 - Scaling
 - Shear
 - Window-to-Viewport
 - Matrices and Vector



Pixel Coordinates

Pixel Coordinates

- A digital image is made up of pixels organized in rows and columns.
- A pixel can be identified by a pair of integers representing its column and row number.
- Columns are typically numbered from left to right starting at zero, while rows may be numbered from top to bottom or bottom to top, starting at zero.
- For example, the pixel with coordinates (3,5) would lie in column number 3 and row number 5.
- In OpenGL, rows are numbered from bottom to top.



12-by-8 pixel grids, shown with row and column numbers.
On the left, rows are numbered from top to bottom,
on the right, they are numbered bottom to top.

Aliasing

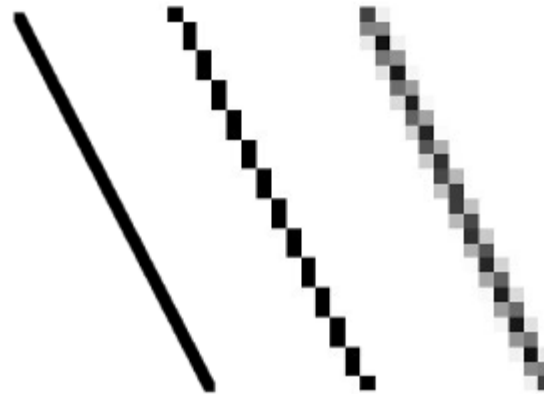
- Row and column numbers identify a pixel, not a point.
- A pixel contains many points; mathematically, it contains an infinite number of points.
- The goal of computer graphics is not really to color pixels - it is to create and manipulate images.
- Ideally, an image should be defined by specifying a color for each point, not just for each pixel. Pixels are an approximation.
- If we imagine that there is a true, ideal image that we want to display, then any image that we display by coloring pixels is an approximation. This has many implications.
- Imagine we want to draw a one-pixel wide line. Unless the line is horizontal or vertical, we can't actually draw the line by coloring pixels.
- It is not possible to make part of a pixel black and part of it white. When we try to draw a line with black and white pixels only, the result is a jagged staircase effect. This effect is an example of something called "**aliasing**."

Antialiasing

- Aliasing can also be seen in the outlines of characters drawn on the screen and in diagonal or curved boundaries between any two regions of different color.
- The term aliasing comes from the fact that ideal images are naturally described in real-number coordinates.
- When we try to represent the image using pixels, many real-number coordinates will map to the same integer pixel coordinates. They can all be considered as different names or "aliases" for the same pixel.
- **Antialiasing techniques** attempt to mitigate the effects of aliasing by mixing the color of a partially covered pixel with the color of the background.
- This is done by calculating the fraction of the pixel that is covered by the shape and using a corresponding shade of gray for the pixel color.
- Calculating this area exactly for each pixel would be too difficult, so some approximate method is used.

Antialiasing

- For example, below is a geometric line, shown on the left, along with two approximations of that line made by coloring pixels.
- The lines are greatly magnified so that we can see the individual pixels. The line on the right is drawn using antialiasing, while the one in the middle is not.
- Note that antialiasing does not give a perfect image, but it can reduce the "jaggedness" that are caused by aliasing.



Pixels and Coordinates – Other Issues

- There are questions about how real-number coordinates are mapped to pixels, such as which point in a pixel corresponds to integer coordinates.
- It is common to consider integer coordinates as referring to the top-left corner of a pixel or to the lines between pixels. This still does not fully determine which pixels are affected when a shape is drawn.
- For example, here are two lines drawn using **HTML canvas graphics**, shown greatly magnified. The lines were specified to be colored black with a one-pixel line width:
- In canvas graphics, integer coordinates correspond to the lines between pixels, and a one-pixel line extends one-half pixel on either side of the geometric line.
- The top line was drawn from (100,100) to (120,100) and extends half into one row of pixels and half into another, which were colored gray using antialiasing
- The bottom line was drawn from (100.5,100.5) to (120.5,100.5) and lies exactly along one line of pixels, which was colored black
- The gray pixels at the ends of the bottom line are due to the fact that the line only extends halfway into the end pixels. Other graphics systems may render these lines differently.



In-Class Demo

The following interactive demo lets you experiment with pixels and antialiasing. Click the question mark icon in the upper-left corner for more information about how to use it.

[Pixel Magnifier Demo](#)



Screen Resolution

- The resolution of a display device is measured in pixels or dots per inch (PPI or DPI).
- Early screens had resolutions of around 72 PPI, while modern high resolution displays can have 200-400 PPI.
- The range of pixel sizes can cause problems when using coordinate systems based on pixels, as an image created assuming a certain PPI may appear too small or too large on a display with a different PPI.
- In many graphics systems, "pixel" is a unit of measure set by the system rather than a physical size, which can change with factors such as magnification.
- On a desktop system, a pixel is usually about one one-hundredth of an inch. On a smart phone, which is usually viewed from a closer distance, the value might be closer to 1/160 inch.
- Vector graphics are less affected by pixels, as the vector image can be created using any convenient coordinate system and the rasterization process of converting it to pixels for display can be left to the hardware.



DOUGLAS COLLEGE

Real-Number Coordinate Systems

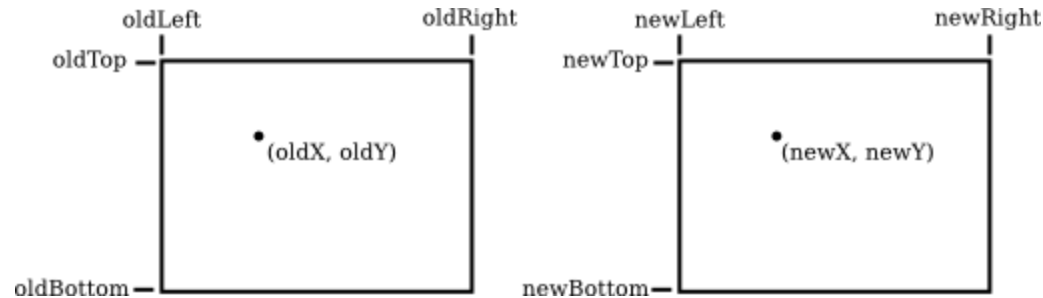
Real-Number Coordinate Systems

- When doing 2D graphics, we are given a rectangle in which to draw primitives using a coordinate system.
- The coordinate system can be chosen based on the application, with a unit of measure and coordinates that are appropriate for the rectangle.
- The coordinate system is specified by the horizontal coordinates for the left and right edges and the vertical coordinates for the top and bottom of the rectangle.
- These values can be labeled as **xmin**, **xmax**, **ymin**, and **ymax**, but it is not necessary to assume that **ymin** is less than **ymax** or that **xmin** is less than **xmax**.
- The vertical coordinate can increase from bottom to top rather than top to bottom, in which case the maximum y-value corresponds to the top of the rectangle.
- To allow programmers to specify the coordinate system that they would like to use, it would be good to have a subroutine such as:

`setCoordinateSystem(left, right, bottom, top)`

Real-Number Coordinate Systems

- The graphics system would then be responsible for automatically transforming the coordinates from the specified coordinate system into pixel coordinates.
- Such a subroutine might not be available, so it's useful to see how the transformation is done by hand.



- Formulas for **newX** and **newY** are then given by:

$$\begin{aligned}\text{newX} &= \text{newLeft} + \\ &\quad ((\text{oldX} - \text{oldLeft}) / (\text{oldRight} - \text{oldLeft})) * (\text{newRight} - \text{newLeft}) \\ \text{newY} &= \text{newTop} + \\ &\quad ((\text{oldY} - \text{oldTop}) / (\text{oldBottom} - \text{oldTop})) * (\text{newBottom} - \text{newTop})\end{aligned}$$

Real-Number Coordinate Systems

- These formulae give **newX** and **newY** as real numbers, and they will have to be rounded or truncated to integer values if we need integer coordinates for pixels.
- The reverse transformation – going from pixel coordinates to real number coordinates – is also useful.
 - For example, if the image is displayed on a computer screen, and we want to react to mouse clicks on the image, we will probably get the mouse coordinates in terms of integer pixel coordinates, but we will want to transform those pixel coordinates into your own chosen coordinate system.
- In practice, we won't usually have to do the transformations ourselves, since most graphics APIs provide some higher level way to specify transforms.



Aspect Ratio

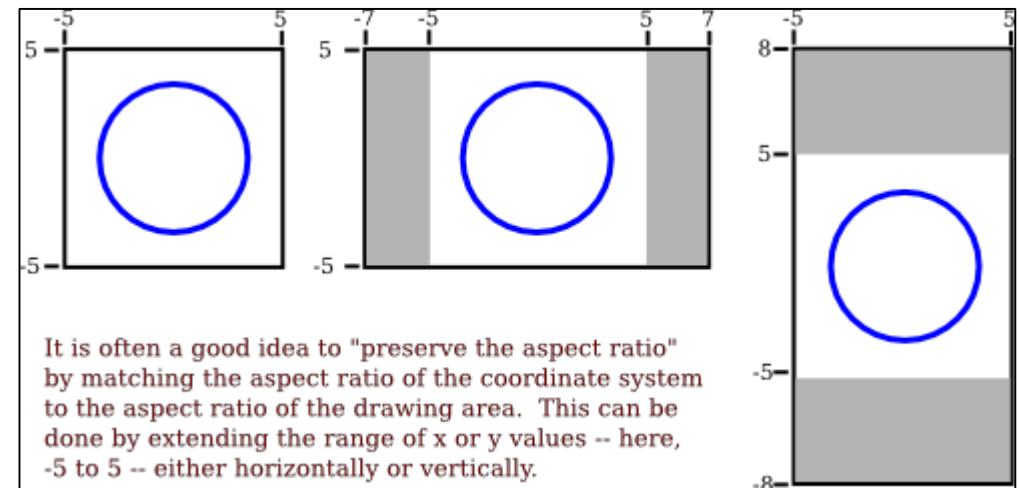
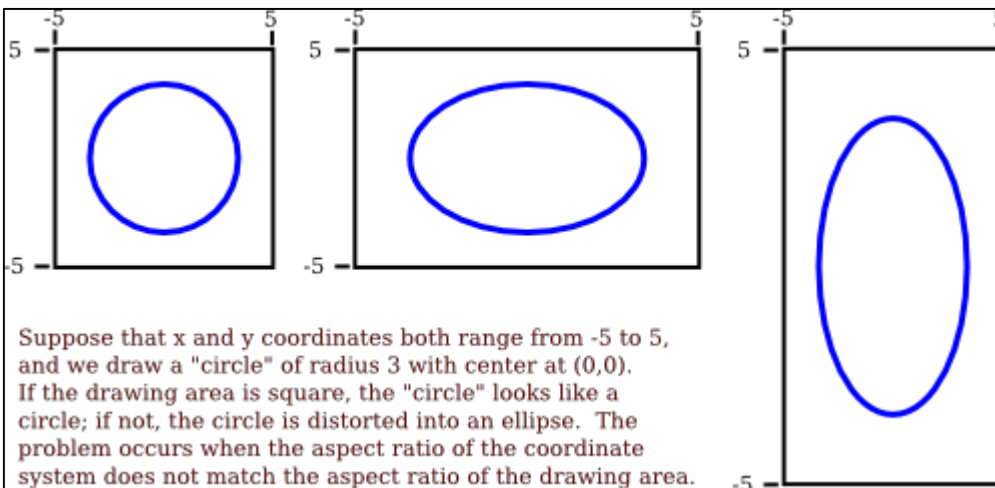
Aspect Ratio

- The aspect ratio of a rectangle is the ratio of its width to its height.
 - For example an aspect ratio of 2:1 means that a rectangle is twice as wide as it is tall, and an aspect ratio of 4:3 means that the width is 4/3 times the height.
- Although aspect ratios are often written in the form width:height, we will use the term to refer to the fraction width/height.
- A square has aspect ratio equal to 1. A rectangle with aspect ratio 5/4 and height 600 has a width equal to $600 \cdot (5/4)$, or 750.
- A coordinate system also has an aspect ratio. If the horizontal and vertical limits for the coordinate system are left, right, bottom, and top, as above, then the aspect ratio is the absolute value of:

$$(\text{right} - \text{left}) / (\text{top} - \text{bottom})$$

Aspect Ratio

- If the coordinate system is used on a rectangle with the same aspect ratio, then when viewed in that rectangle, one unit in the horizontal direction will have the same apparent length as a unit in the vertical direction.
- If the aspect ratios don't match, then there will be some distortion.
- For example, the shape defined by the equation $x^2 + y^2 = 9$ should be a circle, but that will only be true if the aspect ratio of the (x,y) coordinate system matches the aspect ratio of the drawing area.



Color Models

Color Models – RGB

- In the **Red/Green/Blue (RGB) color model**, colors are represented by a combination of red, green, and blue light.
- The intensity of each color is specified by a number between 0 (minimum intensity) and 1 (maximum intensity).
 - For example, the color (1, 0.5, 0.5) is produced by setting red to full intensity, green and blue to half intensity.
- The values for red, green, and blue are called the color components of that color in the RGB model.
- Light is composed of waves with different wavelengths. A pure color is one in which all the light has the same wavelength, but typically a color is made up of many wavelengths - mathematically, an infinite number.
- However, it is not possible to represent all colors by combining just red, green, and blue light.

Color Perception by the Human Eye

- Combinations of the three primary colors (red, green, and blue) are thought to be sufficient to represent all colors since the human eye has three types of color sensors, called cone cells, that detect these colors.
- However, this is only an approximation as cone cells do not exclusively respond to red, green, and blue light.
- Each type of cone cell responds to a range of wavelengths, and the intensity of stimulation determines the color that is perceived.
- A different mix of wavelengths that stimulates each type of cone cell to the same extent will be perceived as the same color.
- A perceived color can be specified by three numbers representing the intensity of stimulation of the three types of cone cells.
- It is not possible to produce all possible patterns of stimulation using just three basic colors.
- Three basic colors can produce a large number of perceivable colors, but there are some colors that cannot be displayed on a computer screen.
- This discussion only applies to people who have three types of cone cells; color blindness, in which an individual is missing one or more types of cone cells, is common.

Color Models – CMYK

- The range of colors that can be produced by a device such as a computer screen is called the **color gamut**.
- Different screens and printers can have different color gamuts.
- The color gamut of a printer is likely smaller than that of a screen, which is why a printed image may not look exactly the same as it did on the screen.
- Printers use a different method to produce colors than screens – they combine inks or dyes instead of light.
- A common color model for printers is **CMYK (cyan, magenta, yellow, and black)**.
- The most common color model for computer graphics is RGB. RGB colors are typically represented using 8 bits per color component, a total of 24 bits per color (also known as 24-bit color).
- An 8-bit number can represent 256 different values, ranging from 0 to 255. A color is specified as a triple of integers (r, g, b) in the range from 0 to 255.

Color Models

- The 8-bit representation of RGB colors works well because it allows for 256 shades of red, green, and blue, which is about as many as the eye can distinguish.
- In some applications, more bits per color component may be used to avoid visual artifacts due to rounding errors in computations.
- These applications may use 16-bit integers or 32-bit floating point values for each color component.
- In other cases, fewer bits may be used, such as a common color scheme that uses 5 bits for red and blue and 6 bits for green (16-bit color) in order to save memory.

Color Models

- The 8-bit representation of RGB colors works well because it allows for 256 shades of red, green, and blue, which is about as many as the eye can distinguish.
- In some applications, more bits per color component may be used to avoid visual artifacts due to rounding errors in computations.
- These applications may use 16-bit integers or 32-bit floating point values for each color component.
- In other cases, fewer bits may be used, such as a common color scheme that uses 5 bits for red and blue and **6 bits for green** (16-bit color) in order to save memory.



Color Models

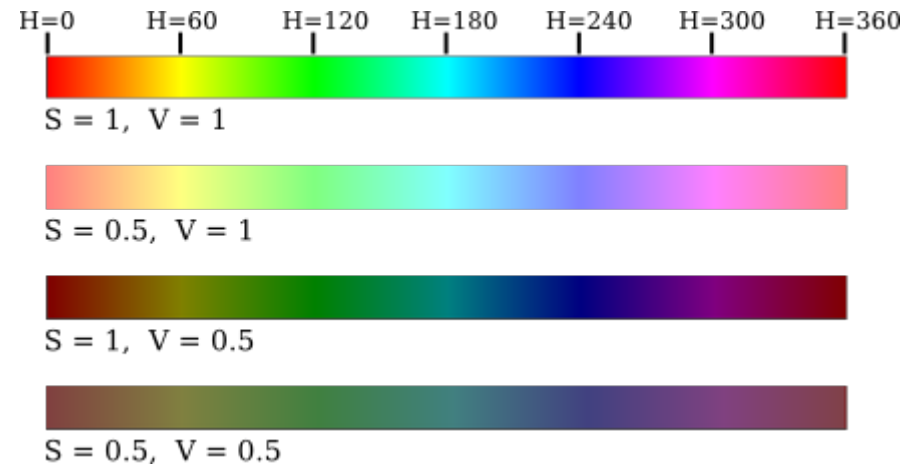
- The 8-bit representation of RGB colors works well because it allows for 256 shades of red, green, and blue, which is about as many as the eye can distinguish.
- In some applications, more bits per color component may be used to avoid visual artifacts due to rounding errors in computations.
- These applications may use 16-bit integers or 32-bit floating point values for each color component.
- In other cases, fewer bits may be used, such as a common color scheme that uses 5 bits for red and blue and 6 bits for green (16-bit color) in order to save memory.
- Green gets an extra bit because the eye is more sensitive to green light than to red or blue.
- The use of fewer bits per color component can result in a loss of color accuracy, but was more common when memory was more expensive.

Color Models – HSV and HSL

- RGB is sometimes criticized for being unintuitive, as it is not obvious that yellow is made of a combination of red and green.
- The **HSV (Hue/Saturation/Value)** and **HSL (Hue/Saturation/Lightness)** color models describe the same set of colors as RGB, but in a more intuitive way.
- HSV is sometimes referred to as HSB, with the "B" standing for "brightness". HSV and HSB are the same model.
- In the HSV and HSL color models, the "H" stands for "hue," which is a basic spectral color. As the value of H increases, the color changes from red to yellow to green to cyan to blue to magenta and then back to red.
- The value of H is often taken to range from 0 to 360.
- The "S" in HSV and HSL stands for "saturation," which is taken to range from 0 to 1. A saturation of 0 results in a shade of gray, while a saturation of 1 gives a "pure color".
- Decreasing the saturation is like adding more gray to the color.
- In the HSV model, "V" stands for "value," which determines the brightness of the color.
- In the HSL model, "L" stands for "lightness," which also determines the brightness of the color.
- The main difference between the two models is that in HSV, the pure spectral colors occur for $V=1$, while in HSL, they occur for $L=0.5$.

Color Models – HSV and HSL

- The illustration below shows colors with a full range of H-values, for S and V equal to 1 and to 0.5.
- Note that for $S=V=1$, we get bright, pure colors. $S=0.5$ gives paler, less saturated colors. $V=0.5$ gives darker colors.



In-Class Demo

The following interactive demo lets you experiment with color creation using the RGB and HSV color models.

[RGB/HSV Color Demo](#)



Color Transparency

- A fourth component called **alpha** is often added to color models. Alpha is used to represent transparency in color models such as **RGBA** and **HSLA**.
- An alpha value of 1 indicates a fully opaque color, while an alpha value of 0 indicates a fully transparent color. Intermediate alpha values produce translucent or partly transparent colors.
- Transparency determines how a foreground color is drawn on top of a background color.
- If the foreground color is fully opaque, it will replace the background color. If the foreground color is partly transparent, it will be blended with the background color.
- The resulting color can be calculated based on the alpha value, which determines the blend between the foreground and background colors:

$$\text{new\_color} = (\text{alpha}) * (\text{foreground\_color}) + (1 - \text{alpha}) * (\text{background\_color})$$

- A 32-bit integer value can be interpreted as a 32-bit RGBA color. Store the alpha component in the eight high-order bits, followed by red, green, and blue.

Shapes (Instead of Pixels!)

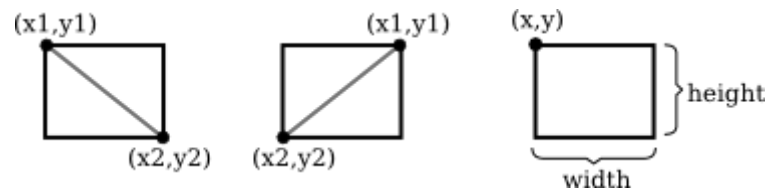
Basic Shapes: Line

- In a graphics API, basic shapes can be drawn with one command, while more complex shapes require multiple commands.
- The basic shapes vary depending on the API, but may include points, lines, and triangles in WebGL.
- Lines, rectangles, and ovals are often considered basic shapes. A line is a straight line segment connecting two points in the plane.
- Lines can be drawn by coloring pixels along the geometric line segment.
- [Bresenham's algorithm](#) is an efficient way to draw lines, but lines can also be complicated by factors such as antialiasing and line width. A wide line may be drawn as a rectangle.
- Lines can have attributes or properties that affect their appearance. One attribute to consider is the appearance at the end of a wide line, which can be a rounded or square cap.
- Another attribute is how lines should be joined when they meet as part of a larger shape.
- Some graphics systems support lines with patterns of dashes and dots.



Basic Shapes: Rectangle

- The basic rectangular shape has sides that are vertical and horizontal (a tilted rectangle requires a rotation).
- A rectangle can be specified by two points, (x_1, y_1) and (x_2, y_2) , that give the endpoints of one of the diagonals, or by a base point, width, and height.
- The width and height must be positive for the rectangle to be non-empty.
- The base point (x, y) determines the corner of the rectangle, with y increasing from top to bottom or from bottom to top depending on the orientation of the coordinate system.



- A common variation on rectangles is to allow rounded corners. For a "round rectangle," the corners are replaced by elliptical arcs.

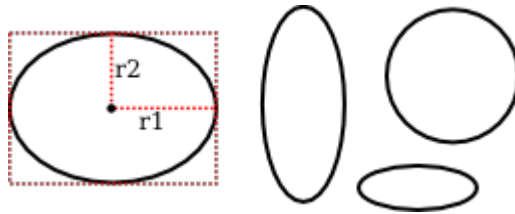


Basic Shapes: Ellipse (or Oval)

- An oval is a closed curve with two radii (vertical and horizontal for a basic oval).
- An oval can be specified by a containing rectangle or by a center point and the lengths of the vertical and horizontal radii.
- If ovals are not available as basic shapes, they can be approximated by drawing a large number of line segments.
- The number of lines that is needed for a good approximation depends on the size of the oval. Suppose that an oval has center point (x,y), horizontal radius r1, and vertical radius r2. Mathematically, the points on the oval are given by:

$$(x + r1*\cos(\text{angle}), y + r2*\sin(\text{angle}))$$

- where angle takes on values from 0 to 360 if angles are measured in degrees or from 0 to 2π if they are measured in radians.
- To get an approximation for an oval, we can use this formula to generate some number of points and then connect those points with line segments.



Approximating an Oval

Draw Oval with center (x,y), horizontal radius r1, and vertical radius r2:

```
for i = 0 to numberOfLines:  
    angle1 = i * (2*pi/numberOfLines)  
    angle2 = (i+1) * (2*pi/numberOfLines)  
    a1 = x + r1*cos(angle1)  
    b1 = y + r2*sin(angle1)  
    a2 = x + r1*cos(angle2)  
    b2 = y + r2*sin(angle2)  
    Draw Line from (a1,b1) to (a2,b2)
```

In-Class Demo

The following interactive demo lets you experiment with using line segments to approximate ovals.

[Approximating an Oval with Line Segments](#)

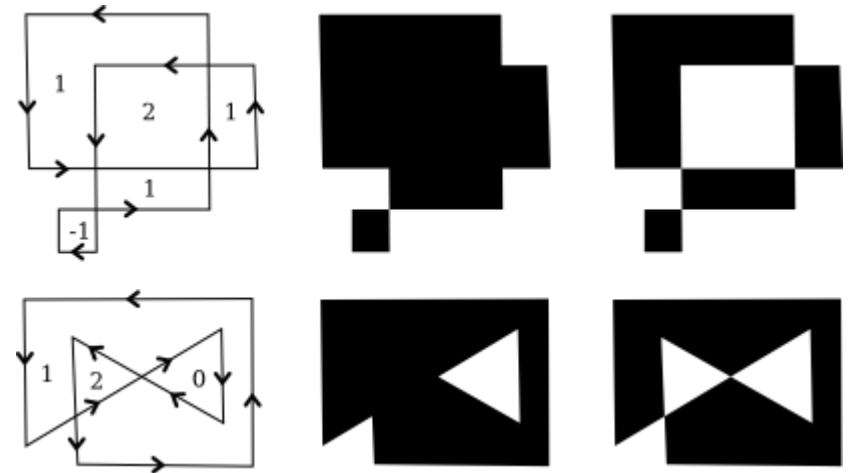


Stroke and Fill



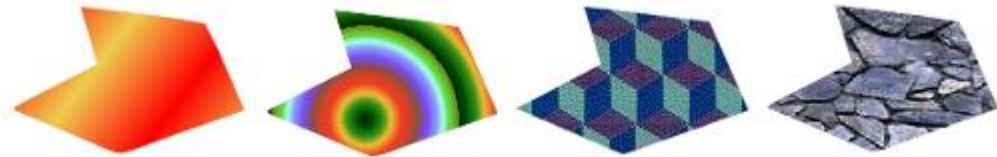
Stroke and Fill

- A shape can be made visible in a drawing by stroking it (drawing the outline) or filling it (coloring the inside).
- It is possible to stroke and fill a shape, giving the interior and outline different appearances.
- When a shape intersects itself, the **winding number** (the number of times it winds around a point in the positive direction) determines how it is filled.
- The winding number can be positive (counterclockwise) or negative (clockwise).
- For the shapes in the center, the fill rule is to color any region that has a non-zero winding number.
- For the shapes shown on the right, the rule is to color any region whose winding number is odd; regions with even winding number are not filled.
- There is still the question of what a shape should be filled with.



Stroke and Fill

- A shape can be filled with a color, pattern, or gradient.
- A **pattern** is a repeating image that covers the shape.
- A **gradient** is a way for color to vary from point to point, computed from endpoints of a line segment and interpolated or extrapolated between them.
- There are different types of gradients, including linear gradients where color is constant along lines perpendicular to the basic line segment and radial gradients where color is constant along circles centered at one endpoint of the line segment.
- Here, the first shape is filled with a simple linear gradient defined by just two colors, while the second shape uses a radial gradient.



- A string of text can be considered to be a shape for the purpose of drawing it. The boundary of the shape is the outline of the characters. The text is drawn by filling that shape.

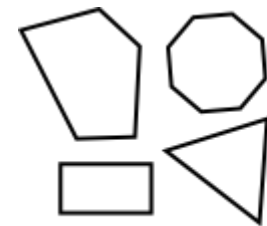


DOUGLAS COLLEGE

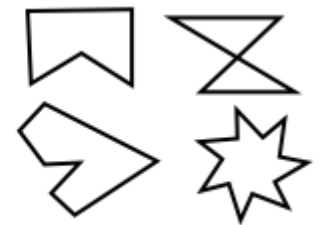
Polygons, Curves, and Paths

Polygons

- Graphics API typically do not include every possible shape as a basic shape. However, there is usually a way to create more complex shapes, such as **polygons**.
- Polygons are a closed shape consisting of connected line segments, with the endpoints being called vertices.
- A **regular polygon** has equal side lengths and angles between sides, with squares and equilateral triangles being examples.
- A **convex polygon** has the property that any two points inside or on the polygon form a line segment that is also entirely inside or on the polygon.
- A convex polygon has no "indentations" along its boundary. **Concavity** can be a property of any shape, not just polygons.
- Sometimes, polygons are required to be "simple," meaning that the polygon has no self-intersections.
- Polygons are usually required to be **planar** as well.



Convex Polygons



Non-convex Polygons

Drawing Polygons

- One method to draw polygons is to use commands for stroking and filling polygons with vertices provided as points or x-y coordinates, but is there a better way?
- Java, SVG, and the HTML canvas API also support the use of a "**path**" to draw shapes.
- A **path** is a general shape that can include line and curved segments and does not have to be connected at endpoints. Paths are better because they allow fills and strokes.
- A path is created by giving a series of commands that specify how a pen would move to draw the shape.
- A point representing the pen's current location is used while creating the path.
- There are commands for moving the pen without drawing and for drawing various types of segments.
- Suppose that we want a path that represents the triangle with vertices at (100,100), (300,100), and (200, 200):

```
createPath()  
moveTo( 100, 100 )  
lineTo( 300, 100 )  
lineTo( 200, 200 )  
closePath()
```

Drawing Polygons – Paths

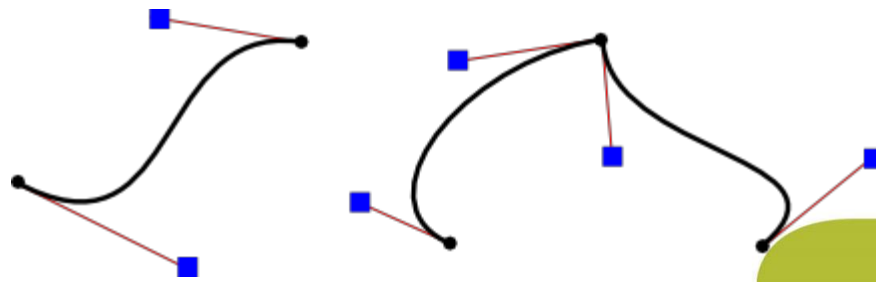
- Earlier, we saw how to approximate an oval by drawing, in effect, a polygon with a large number of sides.
- We drew each side as a separate line segment, so we really had a bunch of separate lines rather than a polygon, but there is no way to fill it.
- It would be better to approximate the oval with a polygonal path. For an oval with center (x, y) and radii $r1$ and $r2$:

```
createPath()  
moveTo( x + r1, y )  
for i = 1 to numberOfPoints-1  
    angle = i * (2*pi/numberOfLines)  
    lineTo( x + r1*cos(angle), y + r2*sin(angle) )  
closePath()
```

- With a path, the computer knows that the sides are part of single shape. This makes it possible to control the appearance of the "join" between consecutive sides.

Bezier Curves

- A path can contain other types of segments besides lines, such as an arc of a circle.
- **Bezier curves** can be used to create general curved shapes and are often used in programs that allow users to design curves interactively.
- Bezier curves are defined by parametric polynomial equations.
- There are two types of Bezier curves: **cubic** and **quadratic**, with cubic being the more common type.
- A cubic Bezier curve segment is defined by the two endpoints of the segment together with two control points.
- The pen starts at the first endpoint, headed in the direction of the first control point. The distance of the control point from the endpoint controls the speed of the pen as it starts drawing the curve. The second control point controls the direction and speed of the pen as it gets to the second endpoint of the curve. There is a unique cubic curve that satisfies these conditions.



In-Class Demo

The following interactive demo lets you edit cubic Bezier curve segments by dragging their endpoints and control points.

[Cubic Bezier Curve Demo](#)



In-Class Demo

The following interactive demo lets you edit cubic Bezier curve segments by dragging their endpoints and control points.

[Cubic Bezier Curve Demo](#)

[Quadratic Bezier Curve Demo](#)





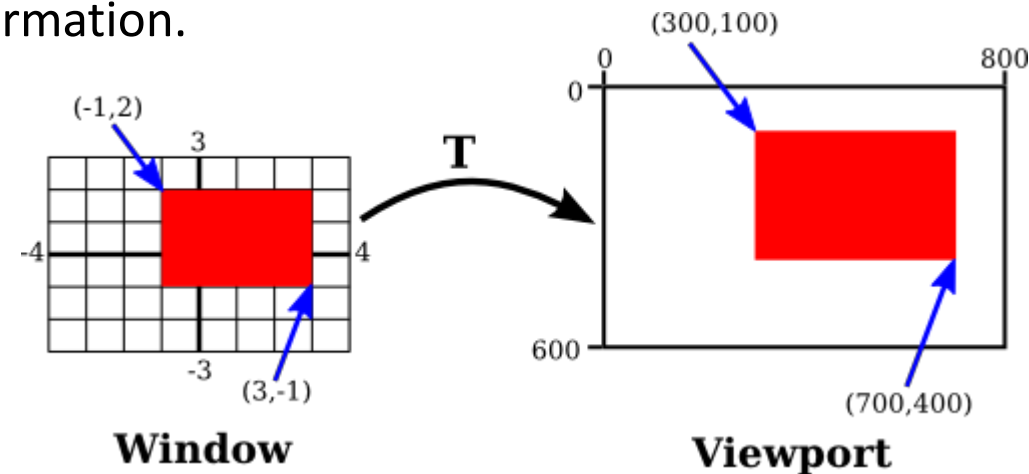
DOUGLAS COLLEGE

Viewing and Modeling

Coordinate Transforms

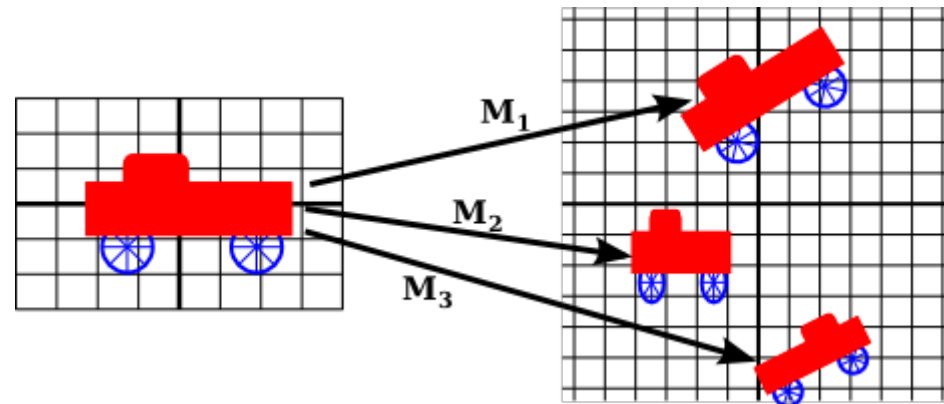
- A rectangle made of pixels called the **viewport** is used to display an image.
- Geometric objects are defined in a different coordinate system using real-number coordinates, called the "**scene**" or "**world**".
- In 2D graphics, the world lies in a plane, and a rectangular area called the window or view window is chosen to display in the image.
- A **coordinate transform** maps the window to the viewport.
- In this illustration, T represents the coordinate transformation.
- T is a function that takes world coordinates (x, y) in some window and maps them to pixel coordinates $T(x, y)$ in the viewport.
- In this example:

$$T(x, y) = (800*(x+4)/8, 600*(3-y)/6)$$



Modeling Transforms

- Complex objects may have multiple copies in a scene, or may need to be placed in different positions in an animation.
- Coordinates used to define an object are called **object coordinates**.
- To place the object in a scene, object coordinates must be transformed into the world coordinate system using a **modeling transformation**.
- Modeling transformations map an object defined in its own object coordinate system into the world coordinate system.
- This diagram illustrates an object defined in its own object coordinate system and then mapped by three different modeling transformations into the world coordinate system:



Modeling Transforms vs. Coordinate Transforms

- Modifying the view window is the same as leaving the coordinate system in place and moving the objects in the scene.
- To get the same effect in the final image, the opposite transformation must be applied to the objects.
- Mathematically, a geometric primitive is specified by giving coordinates in a natural coordinate system and a sequence of transformations is applied to produce the coordinates used to draw the primitive in the image.
- Some transformations may be thought of as modeling transforms and others as coordinate transforms, but to the computer, they are all the same.



In-Class Demo

[Modeling vs. Viewing Demo](#)

Transformation Basics in 2D

- The transforms we will use in 2D graphics can be written in the form:

$$\begin{aligned}x1 &= a*x + b*y + e \\ y1 &= c*x + d*y + f\end{aligned}$$

- where (x, y) represents the coordinates of some point before the transformation is applied, and $(x1, y1)$ are the transformed coordinates.
- The transform is defined by the six constants a, b, c, d, e , and f .
- Note that this can be written as a function T , where

$$T(x, y) = (a*x + b*y + e, c*x + d*y + f)$$

- A transformation of this form is called an **affine transform**. An affine transform has the property that, when it is applied to two parallel lines, the transformed lines will also be parallel.
- Also, if we follow one affine transform by another affine transform, the result is again an affine transform.

Translation

Translation

- A translation transform simply moves every point by a certain amount horizontally and a certain amount vertically.
- If (x, y) is the original point and $(x1, y1)$ is the transformed point, then the formula for a translation is:

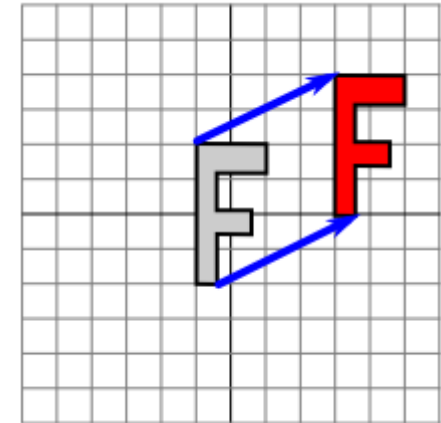
$$x1 = x + e$$

$$y1 = y + f$$

- where e is the number of units by which the point is moved horizontally and f is the amount by which it is moved vertically.
- A 2D graphics system will typically have a function such as:

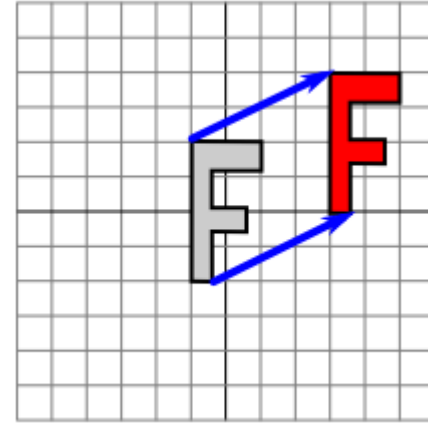
`translate( e, f )`

- The translation would apply to everything that is **drawn after** the command is given. That is, for all subsequent drawing operations, e would be added to the x-coordinate and f would be added to the y-coordinate.



Translation

- In this diagram, the light gray "F" shows what would be drawn without the translation; the dark red "F" shows the same "F" drawn after applying a translation by (4,2).
- The top arrow shows that the upper left corner of the "F" has been moved over 4 units and up 2 units.
- When we give the command `translate(e, f)`, the translation applies to all the drawing that follows, not just to the next shape.
- If we apply another transformation after the translation, the second transform will **not** replace the translation. It will be combined with the translation, so that subsequent drawing will be affected by the combined transformation.
 - For example, if we combine `translate(4, 2)` with `translate(-1, 5)`, the result is the same as a single translation, `translate(3, 7)`.





DOUGLAS COLLEGE

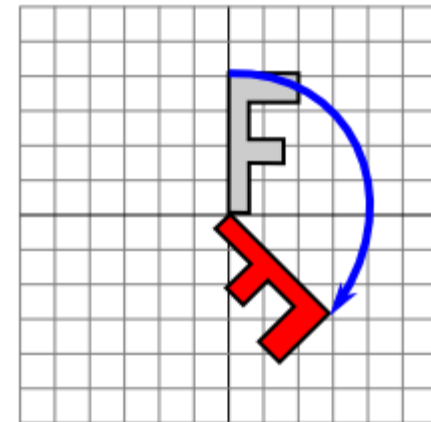
Rotation

Rotation

- A **rotation transform** rotates each point about the origin by a specified angle of rotation.
- Angles can be measured in degrees or radians (Java and JavaScript use radians, while OpenGL and SVG use degrees).
- A positive angle rotates objects counterclockwise in a coordinate system where the y-axis points up and clockwise in the usual pixel coordinates where the y-axis points down.
- When rotation through an angle of r radians about the origin is applied to the point (x, y) , then the resulting point (x_1, y_1) is given by:

$$\begin{aligned}x_1 &= \cos(r) * x - \sin(r) * y \\y_1 &= \sin(r) * x + \cos(r) * y\end{aligned}$$

- A 2D graphics API would typically have a command `rotate(r)` to apply a rotation. The command is used before drawing the objects to which the rotation applies.





DOUGLAS COLLEGE

Combining Transformations

Combining Transforms

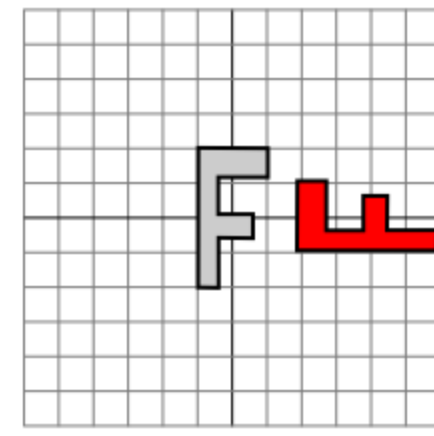
- Suppose that before drawing some object, we write:

```
translate(4,0)  
rotate(90)
```

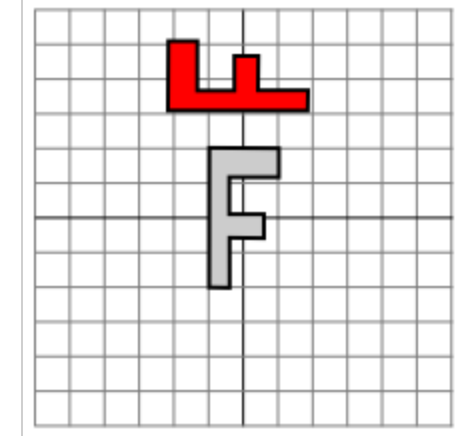
- The translation will then apply to all subsequent drawing. But, because of the rotation command, the things drawn after the translation are rotated objects (i.e., the translation applies to objects that have already been rotated).
- If we reverse the order in which the two transforms are applied in this example, by writing:

```
rotate(90)  
translate(4,0)
```

- then the result is as shown on the right in the diagram.



Rotate then translate



Translate then rotate

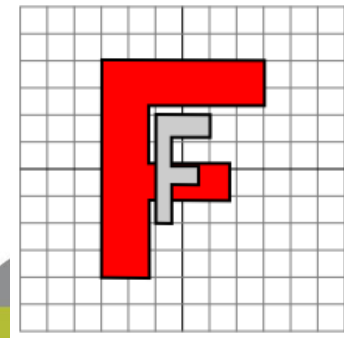


DOUGLAS COLLEGE

Scaling

Scaling

- A **scaling transform** can be used to make objects bigger or smaller.
- if a point (x, y) is scaled by a factor of a in the x direction and by a factor of d in the y direction, then the resulting point $(x1, y1)$ is given by:
$$x1 = a * x$$
$$y1 = d * y$$
- The common case where the horizontal and vertical scaling factors are the same is called **uniform scaling**. Uniform scaling stretches or shrinks a shape without distorting it.
- A 2D graphics API can provide a function `scale(a, d)` for applying scaling transformations.
- Negative scaling factors are allowed and will result in reflecting the shape as well as possibly stretching or shrinking it. For example, `scale(1, -1)` will reflect objects vertically, through the x -axis.
- **Every affine transform can be created by combining translations, rotations about the origin, and scalings about the origin.**



In-Class Demo

The following interactive demo lets you experiment with translations, rotations, and scalings, and with the transformations that can be made by combining them.

[2D Transforms](#)

A transform that is made from translations and rotations, with no scaling, will preserve length and angles in the objects to which it is applied. It will also preserve aspect ratios of rectangles.

Transforms with this property are called "**Euclidean**." If we also allow uniform scaling, the resulting transformation will preserve angles and aspect ratio, but not lengths.



Shearing Transform

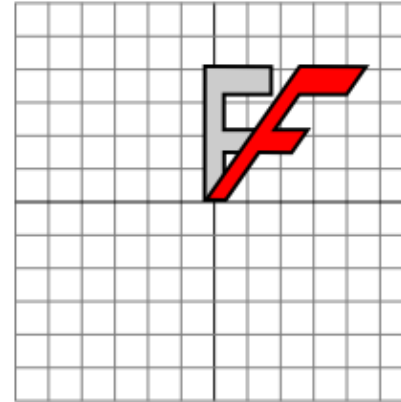
Shear

- A **shearing transform** tilts objects.
- A horizontal shear tilts objects towards the left (for negative) or right (for positive).
- A vertical shear tilts objects up or down.
- Here is an example of horizontal shear:
- A horizontal shear does not move the x-axis. Every other horizontal line is moved to the left or to the right by an amount that is proportional to the y-value along that line.
- When a horizontal shear is applied to a point (x, y) , the resulting point $(x1, y1)$ is given by:

$$\begin{aligned}x1 &= x + b * y \\ y1 &= y\end{aligned}$$

- Similarly, a vertical shear is given by:

$$\begin{aligned}x1 &= x \\ y1 &= c * x + y\end{aligned}$$





DOUGLAS COLLEGE

Window-to-View Transform

Window-to-View Transform

- The last transformation applied to an object before it is displayed in an image is the **window-to-viewport transformation**, which maps the view window containing the scene to the grid of pixels where the image will be displayed.
- In the case where the view window is not rotated and has sides parallel to the x- and y-axes, the window-to-viewport transformation can be expressed using translation and scaling transforms.
- Assume the viewport has pixel coordinates ranging from 0 on the left to **width** on the right and from 0 at the top to **height** at the bottom.
- And assume that the limits on the view window are **left**, **right**, **bottom**, and **top**.
- The window-to-viewport transformation can be programmed as:

```
scale( width / (right-left), height / (bottom-top) );  
translate( -left, -top )
```


Window-to-View Transform

```
scale( width / (right-left), height / (bottom-top) );  
translate( -left, -top )
```

- These should be the last transforms that are applied to a point.
- Since **transforms are applied to points in the reverse of the order in which they are specified in the program**, they should be the first transforms specified in the program.
- Consider a point (x, y) in the view window.
- The coordinates (x, y) are first translated by $(-left, -top)$ to give $(x-left, y-top)$. These coordinates are then multiplied by the scaling factors shown above, giving the final coordinates:

$$\begin{aligned}x1 &= \text{width} / (\text{right-left}) * (x-left) \\ y1 &= \text{height} / (\text{bottom-top}) * (y-top)\end{aligned}$$

- Note that the point $(left, top)$ is mapped to $(0, 0)$, while the point $(right, bottom)$ is mapped to $(width, height)$, exactly what we want!

Window-to-View Transform

- If we want to force the aspect ratio of the window to match the aspect ratio of the viewport, it might be necessary to adjust the limits on the window.

```
subroutine applyWindowToViewportTransformation (
    left, right,    // horizontal limits on view window
    bottom, top,    // vertical limits on view window
    width, height, // width and height of viewport
    preserveAspect // should window be forced to match viewport aspect?
)
```

```
if preserveAspect :
    // Adjust the limits to match the aspect ratio of the drawing area.
    displayAspect = abs(height / width);
    windowAspect = abs(( top-bottom ) / ( right-left ));
    if displayAspect > windowAspect :
        // Expand the viewport vertically.
        excess = (top-bottom) * (displayAspect/windowAspect - 1)
        top = top + excess/2
        bottom = bottom - excess/2
    else if displayAspect < windowAspect :
        // Expand the viewport horizontally.
        excess = (right-left) * (windowAspect/displayAspect - 1)
        right = right + excess/2
        left = left - excess/2
```

```
scale( width / (right-left), height / (bottom-top) )
translate( -left, -top )
```

Matrices and Vectors

Matrices and Vectors

- Transforms used in computer graphics can be represented as matrices and points as vectors.
- Matrices are two-dimensional arrays of numbers and vectors are one-dimensional arrays.
- Linear algebra is essential to computer graphics and matrix and vector math is built into GPUs.
- Vectors are lists of 2, 3, or 4 numbers and are often written as (x, y) , (x, y, z) , and (x, y, z, w) .
- Matrices with N rows and M columns are called " N -by- M matrices", and the matrices needed in computer graphics are usually N -by- N with $N = 2, 3$, or 4 .
- If A and B are two N -by- N matrices, they can be multiplied to give a product matrix $C = AB$.
- If A is an N -by- N matrix and v is a vector of length N , v can be multiplied by A to give another vector $w = Av$.
- The function that takes v to Av is a **linear transformation**.
- Rotation and scaling are linear transformations that can be represented as matrices.

Matrices and Vectors

- The operation of rotating (x,y) through an angle d about the origin can be done by multiplying (x,y) by a 2-by-2 matrix, R_d .
- Similarly, scaling by a factor a in the horizontal direction and b in the vertical direction can be given as a matrix $S_{a,b}$.
- If we want to apply a scaling followed by a rotation to the point $v = (x,y)$, we can compute either $R_d(S_{a,b}v)$ or $(R_dS_{a,b})v$.
- Suppose that we want to apply the same two operations, scale then rotate, to thousands of points, as we typically do when transforming objects for computer graphics.
- We could compute the product matrix $R_dS_{a,b}$ once and for all, and then apply the combined transform to each point with a single multiplication!
- If a program says
 $\text{rotate}(d)$ 50 times, the computer **combines the operations into a**
 $\text{scale}(a,b)$
single matrix and just keeps track of that. **Multiple transformations can be handled as efficiently as a single transformation!**

Matrices and Vectors



- **Translation is not a linear transformation.**
- To bring translation into this framework, instead of representing a point in 2D as a pair of numbers (x, y) , we represent it as the triple of numbers $(x, y, 1)$.
- It turns out that we can then represent rotation, scaling, and translation – **and hence any affine transformation** – on 2D space as multiplication by a 3-by-3 matrix.
- The matrices that we need have a bottom row containing $(0,0,1)$. Multiplying $(x,y,1)$ by such a matrix gives a new vector $(x1,y1,1)$. We ignore the extra coordinate and consider this to be a transformation of (x,y) into $(x1,y1)$.
- The 3-by-3 matrices for translation ($T_{a,b}$), scaling ($S_{a,b}$), and rotation (R_d) in 2D are:

$$T_{a,b} = \begin{bmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{bmatrix}$$

$$S_{a,b} = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R_d = \begin{bmatrix} \cos d & -\sin d & 0 \\ \sin d & \cos d & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- A sequence of transformations can be combined into a single transformation. The computer only needs to keep track of a single matrix, which we call the "**current matrix**" or "**current transformation**."

References

- Interactive Computer Graphics, a Top-Down Approach with WebGL, 7th Edition – Edward Angel and Dave Shreiner, Pearson.
- David J. Eck, Introduction to Computer Graphics, Version 1.3, August 2021. Notes from this source are licensed under a [Creative Commons Attribution-Noncommercial-ShareAlike 4.0 License](https://creativecommons.org/licenses/by-nc-sa/4.0/).



Thank you.
DOUGLASCOLLEGE

DOUGLASCOLLEGE

CMPT 2361

Computer Graphics

Shahriar Khosravi, Ph.D.

Lecture 3

Plan for this Lecture

- HTML Canvas Graphics
 - The 2D Graphics Context
 - Shapes
 - Stroke and Fill
 - Auxiliary Canvases
 - Pixel Manipulation
 - Images



DOUGLAS COLLEGE

The 2D Graphics Context

HTML Canvas Graphics

- The visible content of a web page is made up of "**elements**" such as headlines and paragraphs.
- The content is specified using the HTML language.
- A "**canvas**" is an HTML element. It appears on the page as a blank rectangular area which can be used as a drawing surface by the "HTML canvas" graphics API.
- In the source code of a web page, a canvas element is created with code of the form:

```
<canvas width="800" height="600" id="theCanvas"></canvas>
```
- To draw on a canvas, we need a **graphics context**, which is an object that contains functions for drawing shapes.
- It also contains variables that record the current graphics state, including the current drawing color, transform, and font.
- Usually, we use **graphics** as the name of the variable that refers to the graphics context, but the variable name is up to the programmer.

HTML Canvas Graphics

- A typical starting point is:

```
canvas = document.getElementById("theCanvas");  
graphics = canvas.getContext("2d");
```

gets a reference to the canvas element on the web page, using its id.

creates the graphics context for that canvas element.

- This code will produce an error in a web browser that doesn't support canvas, so we might add error checking by putting these commands inside a try-catch statement.
- Typically, we store the canvas graphics context in a global variable and use the same graphics context throughout our program.
- When a graphics context is global, changes made to the state in one function call will carry over to subsequent function calls, unless we do something to limit their effect.
- This can actually lead to a common type of bug.
 - For example, if we apply a 30-degree rotation in a function, those rotations will accumulate each time the function is called, unless we do something to undo the previous rotation before the function is called again.

HTML Canvas Graphics

- Here is the source code for a very minimal web page that uses canvas graphics:

```
<!DOCTYPE html>
<html>
<head>
<title>Canvas Graphics</title>
<script>
    let canvas;    // DOM object corresponding to the canvas
    let graphics;  // 2D graphics context for drawing on the canvas

    function draw() {
        // draw on the canvas, using the graphics context
        graphics.fillText("Hello World", 10, 20);
    }

    function init() {
        canvas = document.getElementById("theCanvas");
        graphics = canvas.getContext("2d");
        draw(); // draw something on the canvas
    }

    window.onload = init;

</script>
</head>
<body>
    <canvas id="theCanvas" width="640" height="480"></canvas>
</body>
</html>
```



DOUGLAS COLLEGE

Shapes

Shapes

- The default coordinate system on a canvas has (0,0) at the upper left corner, x-coordinates increasing to the right, and y-coordinates increasing downward.
- The unit of measure is nominal pixel, which may correspond to multiple actual pixels on high-resolution screens.
- The range of x and y values is determined by the width and height properties of the `<canvas>` element.
- Applying magnification to a browser window will stretch the unit of measure.
- The canvas API only supports basic shapes such as rectangles and text.
- Other shapes must be created as paths.
- Shapes can be stroked and filled, and this includes text.
- Stroking text is only effective when the characters are large.

Shapes

- Here are the functions for drawing rectangles and text, where **graphics** refers to the object that represents the graphics context.
- **graphics.fillRect(x,y,w,h)** – draws a filled rectangle with corner at (x,y), with width w and with height h. If the width or the height is less than or equal to zero, nothing is drawn.
- **graphics.strokeRect(x,y,w,h)** – strokes the outline of the same rectangle.
- **graphics.clearRect(x,y,w,h)** – clears the rectangle by filling it with fully transparent pixels, allowing the background of the canvas to show. The background is determined by the properties of the web page on which the canvas appears. It might be a background color, an image, or even another canvas.
- **graphics.fillText(str,x,y)** – fills the characters in the string **str**. The left end of the baseline of the string is positioned at the point (x,y).
- **graphics.strokeText(str,x,y)** – strokes the outlines of the characters in the string.

Paths

- A **path** can be created using functions in the graphics context.
- The context keeps track of a "current path."
- In the current version of the API, paths are not represented by objects, and there is no way to work with more than one path at a time or to keep a copy of a path for later reuse.
- Paths can contain lines, Bezier curves, and circular arcs.
- Here is an overview of the most common functions for working with paths.
- `graphics.beginPath()` – start a new path. Any previous path is discarded, and the current path in the graphics context is now empty. Note that the graphics context also keeps track of the current point, the last point in the current path. After calling `graphics.beginPath()`, the current point is undefined.
- `graphics.moveTo(x,y)` – move the current point to `(x,y)`, without adding anything to the path. This can be used for the starting point of the path or to start a new, disconnected segment of the path.
- `graphics.lineTo(x,y)` – add the line segment starting at current point and ending at `(x,y)` to the path, and move the current point to `(x,y)`.

Paths

- `graphics.bezierCurveTo(cx1, cy1, cx2, cy2, x, y)` – add a cubic Bezier curve to the path. The curve starts at the current point and ends at (x, y) . The points $(cx1, cy1)$ and $(cx2, cy2)$ are the two control points for the curve.
- `graphics.quadraticCurveTo(cx, cy, x, y)` – adds a quadratic Bezier curve from the current point to (x, y) , with control point (cx, cy) .
- `graphics.arc(x, y, r, startAngle, endAngle)` – adds an arc of the circle with center (x, y) and radius r . The next two parameters give the starting and ending angle of the arc. They are measured in radians. The positive rotation direction is clockwise in the default coordinate system. The arc extends in the positive direction from the start angle to the end angle. An optional fifth parameter can be set to true to get an arc that extends in the negative direction. After drawing the arc, the current point is at the end of the arc. If there is a current point before `graphics.arc` is called, then before the arc is drawn, a line is added to the path that extends from the current point to the starting point of the arc. Recall that immediately after `graphics.beginPath()`, there is no current point.
- `graphics.closePath()` – adds to the path a line from the current point back to the starting point of the current segment of the curve. (Recall that you start a new segment of the curve every time you use `moveTo`.)

Paths

- `graphics.bezierCurveTo(cx1, cy1, cx2, cy2, x, y)` – add a cubic Bezier curve to the path. The curve starts at the current point and ends at (x, y) . The points $(cx1, cy1)$ and $(cx2, cy2)$ are the two control points for the curve.
- `graphics.quadraticCurveTo(cx, cy, x, y)` – add a quadratic Bezier curve from the current point to (x, y) with control point (cx, cy) .
- `graphics.arc(x, y, radius, startAngle, endAngle, clockwise)` – draw the circle with center (x, y) and radius `radius`. The angle `startAngle` is the starting angle of the arc. The angle `endAngle` is the ending angle of the arc. The boolean `clockwise` indicates whether the arc is drawn clockwise in the path. If `clockwise` is `true`, the arc is drawn clockwise from the start angle to the end angle. If `clockwise` is `false`, the arc is drawn counter-clockwise from the start angle to the end angle. If there is a current point before `graphics.arc` is called, then before the arc is drawn, a line is added to the path that extends from the current point to the starting point of the arc. Recall that immediately after `graphics.beginPath()`, there is no current point.
- `graphics.closePath()` – adds to the path a line from the current point back to the starting point of the current segment of the curve. (Recall that you start a new segment of the curve every time you use `moveTo`.)

Creating a curve with these commands does not draw anything. To get something visible to appear in the image, you must fill or stroke the path.

Filling and Stroking Paths

- The commands `graphics.fill()` and `graphics.stroke()` are used to fill and to stroke the current path.
- If we fill a path that has not been closed, the `fill()` algorithm acts as though a final line segment had been added to close the path.
- When we stroke a shape, it's the center of the virtual pen that moves along the path. So, for high-precision canvas drawing, it's common to use paths that pass through the centers of pixels rather than through their corners.
- For example, to draw a line that extends from the pixel with coordinates (100,200) to the pixel with coordinates (300,200), you would actually stroke the geometric line with endpoints (100.5,200.5) and (100.5,300.5).

- Here is an example that draws a line:

```
graphics.beginPath();           // start a new path
graphics.moveTo(100.5,200.5);    // starting point of the new path
graphics.lineTo(300.5,200.5);    // add a line to the point (300.5,200.5)
graphics.stroke();               // draw the line
```

- Remember that the line remains part of the current path until the next time we call `graphics.beginPath()`.

Filling and Stroking Paths

Here's an example to draw a filled, regular octagon centered at (200,400) and with radius 100:

```
graphics.beginPath();
graphics.moveTo(300,400);
for (let i = 1; i < 8; i++) {
  let angle = (2*Math.PI)/8 * i;
  let x = 200 + 100*Math.cos(angle);
  let y = 400 + 100*Math.sin(angle);
  graphics.lineTo(x,y);
}
graphics.closePath();
graphics.fill();
```

Here is how to draw a wedge of a circle:

```
graphics.beginPath();
graphics.moveTo(200,300); // Move current point to center of the circle.
graphics.arc(200,300,100,0,Math.PI/4); // Arc, plus line from current point.
graphics.lineTo(200,300); // Line from end of arc back to center of circle.
graphics.fill(); // Fill the wedge.
```

Here's a filled circle with radius 100, centered at 200,300:

```
graphics.beginPath();
graphics.arc( 200, 300, 100, 0, 2*Math.PI );
graphics.fill();
```

To draw just the outline of the circle, use `graphics.stroke()` in place of `graphics.fill()`. We can apply both operations to the same path.

In-Class Demo

There is no way to draw an oval that is not a circle, except by using transforms, but JavaScript has the interesting property that it is possible to add new functions and properties to an existing object. The following demo shows how to add functions to a graphics context for drawing lines, ovals, and other shapes that are not built into the API.

[Canvas Graphics Starter \(Extra\)](#)



Stroke and Fill



Stroke and Fill

- Attributes such as line width that affect the visual appearance of strokes and fills are stored as properties of the graphics context.
- For example, the value of `graphics.lineWidth` is a number that represents the width that will be used for strokes.
- The width is given in pixels for the default coordinate system, but it is subject to transforms. We can change the line width by assigning a value to this property:

```
graphics.lineWidth = 2.5; // Change the current width.
```

- The change affects subsequent strokes. we can also read the current value:

```
saveWidth = graphics.lineWidth; // Save current width.
```

- The property `graphics.lineCap` controls the appearance of the endpoints of a stroke. It can be set to "round", "square", or "butt". The quotation marks are part of the value.

```
graphics.lineCap = "round";
```

Stroke and Fill

- Similarly, `graphics.lineJoin` controls the appearance of the point where one segment of a stroke joins another segment; its possible values are "round", "bevel", or "miter".
- Color is controlled by the values of the properties `graphics.fillStyle` and `graphics.strokeStyle`.
- The graphics context maintains separate styles for filling and for stroking. A solid color for stroking or filling is specified as a string.
- Valid color strings are ones that can be used in CSS, the language that is used to specify colors and other style properties of elements on web pages.
- Many solid colors can be specified by their names, such as "red", "black", and "beige".
- An RGB color can be specified as a string of the form "rgb(r, g, b)", where the parentheses contain three numbers in the range 0 to 255 giving the red, green, and blue components of the color.
- Hexadecimal color codes are also supported, in the form "#XXYYZZ" where XX, YY, and ZZ are two-digit hexadecimal numbers giving the RGB color components.

```
graphics.fillStyle = "rgb(200,200,255)"; // light blue  
graphics.strokeStyle = "#0070A0"; // a darker, greenish blue
```

Gradient Fill

- Gradients and patterns are also supported. As an example, a gradient can be created with a series of steps:

```
let lineargradient = graphics.createLinearGradient(420,420,550,200);  
lineargradient.addColorStop(0,"red");  
lineargradient.addColorStop(0.5,"yellow");  
lineargradient.addColorStop(1,"green");  
graphics.fillStyle = lineargradient; // Use a gradient fill!
```

- The first line creates a linear gradient that will vary in color along the line segment from the point (420,420) to the point (550,200).
- Colors for the gradient are specified by the `addColorStop()` function: the first parameter gives the fraction of the distance from the initial point to the final point where that color is applied, and the second is a string that specifies the color itself.
- A color stop at 0 specifies the color at the initial point; a color stop at 1 specifies the color at the final point. Once a gradient has been created, it can be used both as a fill style and as a stroke style in the graphics context.

Font Properties

- The font that is used for drawing text is the value of the property `graphics.font`.
- The value is a string that could be used to specify a font in CSS. As such, it can be fairly complicate.

```
graphics.font = "2cm monospace"; // the size is in centimeters
graphics.font = "bold 18px sans-serif";
graphics.font = "italic 150% serif"; // size is 150% of the usual size
```

- Note that text, like all drawing, is subject to coordinate transforms.
- Applying a scaling operation changes the size of the text, and a negative scaling factor can produce mirror-image text.



DOUGLAS COLLEGE

Transforms

Transforms

- A graphics context has three basic functions for modifying the current transform by scaling, rotation, and translation.
- There are also functions that will compose the current transform with an arbitrary transform and for completely replacing the current transform:
- `graphics.scale(sx, sy)` – scale by `sx` in the x-direction and `sy` in the y-direction.
- `graphics.rotate(angle)` – rotate by `angle` radians about the origin. A positive rotation is clockwise in the default coordinate system.
- `graphics.translate(tx, ty)` – translate by `tx` in the x-direction and `ty` in the y-direction.
- `graphics.transform(a, b, c, d, e, f)` – apply the affine transform $x1 = a*x + c*y + e$, and $y1 = b*x + d*y + f$.
- `graphics.setTransform(a, b, c, d, e, f)` – discard the current transformation, and set the current transformation to be $x1 = a*x + c*y + e$, and $y1 = b*x + d*y + f$.

Transforms

- A graphics context has three basic functions for modifying the current transform by scaling, rotation, and translation.
- There are also functions for scaling by arbitrary factors, rotating by arbitrary angles, and translating by arbitrary distances.
- `graphics.transform(sx, sy, tx, ty, angle)` – apply the transform $x1 = sx * x + tx$ and $y1 = sy * y + ty$ to the current transform, and rotate by `angle` radians about the origin. A positive rotation is clockwise in the default coordinate system.
- `graphics.translate(tx, ty)` – translate by `tx` in the x-direction and `ty` in the y-direction.
- `graphics.transform(a, b, c, d, e, f)` – apply the affine transform $x1 = a * x + c * y + e$, and $y1 = b * x + d * y + f$.
- `graphics.setTransform(a, b, c, d, e, f)` – discard the current transformation, and set the current transformation to be $x1 = a * x + c * y + e$, and $y1 = b * x + d * y + f$.

Note that **there is no shear transform**, but you can apply a shear as a general transform. For example, for a horizontal shear with shear factor 0.5, use:

```
graphics.transform(1, 0, 0.5, 1, 0, 0)
```

Transforms

- To implement hierarchical modeling, we need to be able to save the current transformation so that we can restore it later.
- Unfortunately, no way is provided to read the current transformation from a canvas graphics context.
- However, the graphics context itself keeps a stack of transformations and provides methods for pushing and popping the current transformation.
- These methods do more than save and restore the current transformation. They actually save and restore almost the entire state of the graphics context, including properties such as current colors, line width, and font (**but not the current path**):
- `graphics.save()` – push a copy of the current state of the graphics context, including the current transformation, onto the stack.
- `graphics.restore()` – remove the top item from the stack, containing a saved state of the graphics context, and restore the graphics context to that state.

Transforms

- Using these methods, the basic setup for drawing an object with a modeling transform becomes:

```
graphics.save();           // save a copy of the current state
graphics.translate(a,b);   // apply modeling transformations
graphics.rotate(r);
graphics.scale(sx,sy);
.
. // Draw the object!
.
graphics.restore();        // restore the saved state
```

- Note that if drawing the object includes any changes to attributes such as drawing color, those changes will be also undone by the call to **graphics.restore()**.
- In hierarchical graphics, this is what we want, and it eliminates the need to have extra statements for saving and restoring things like color.

Transforms

- Suppose that we want an oval with center at (x, y) , with horizontal radius **r1** and with vertical radius **r2**.
- We would like to draw a circle of radius 1 with center at $(0,0)$, then transform it. The circle needs to be scaled by a factor of **r1** horizontally and **r2** vertically. It should then be translated to move its center from $(0,0)$ to (x, y) .
- We can use `graphics.save()` and `graphics.restore()` to make sure that the transformations only affect the circle.
- Recall that the order of transforms in the code is the opposite of the order in which they are applied to objects.

```
graphics.save();  
graphics.translate( x, y );  
graphics.scale( r1, r2 );  
graphics.beginPath();  
graphics.arc( 0, 0, 1, 0, 2*Math.PI ); // a circle of radius 1  
graphics.restore();  
graphics.stroke();
```

Transforms

- Note that the current path is not affected by the calls to `graphics.save()` and `graphics.restore()`.
- So, the oval-shaped path is not discarded when `graphics.restore()` is called.
- When `graphics.stroke()` is called at the end, it is the oval-shaped path that is stroked.
- On the other hand, the line width that is used for the stroke is not affected by the scale transform that was applied to the oval.
- Note that if the order of the last two commands were reversed, **then the line width would be subject to the scaling.**

```
graphics.save();
graphics.translate( x, y );
graphics.scale( r1, r2 );
graphics.beginPath();
graphics.arc( 0, 0, 1, 0, 2*Math.PI ); // a circle of radius 1
graphics.restore();
graphics.stroke();
```

Transforms

- In the HTML canvas API, the points that are used to create a path are transformed by the current transformation before they are saved. That is, they are saved in **pixel coordinates**.
- Later, when the path is stroked or filled, the current transform has no effect on the path (although it can affect, for example, the line width when the path is stroked).
- In particular, we cannot make a path and then apply different transformations.
- For example, we cannot make an oval-shaped path, and then use it to draw several ovals in different positions.
- Every time we draw the oval, it will be in the same place, even if different translation transforms are applied to the graphics context.
- This is different from other graphics APIs (e.g., Java).
- In Java, the coordinates that are stored in the path are the actual numbers that are used to specify the path, that is, the **object coordinates**.
- when the path is stroked or filled, the transformation that is in effect at that time is applied to the path. The path can be reused many times to draw copies with different transformations.



Auxiliary Canvases

Auxiliary Canvases

- It is possible to work with images that are not visible on the screen, known as an **off-screen canvas**.
- It is useful to be able to quickly copy the off-screen canvas onto the screen.
- One reason for using an off-screen canvas is to be able to make changes to an image without altering the official copy.
- Changes to the image are made to the off-screen canvas, then copied to the screen.
- An example use case is drawing a box around a selected region on the screen without damaging the official copy in the off-screen canvas.
- To remove the box, the off-screen canvas image can be copied onto the screen again.

In-Class Demo

The sample program lets the user drag the mouse on the canvas to draw some shapes. It uses two on-screen canvases, one located exactly on top of the other. The lower canvas holds the actual image. The upper canvas is completely transparent, except when the user is drawing a line, oval, or rectangle. While the user is dragging the mouse to draw such a shape, the new shape is drawn on the upper canvas, where it hides the part of the lower canvas that is beneath the shape. When the user releases the mouse, the shape is added to the lower canvas and the upper canvas is cleared to make it completely transparent again. The other tools operate directly on the lower canvas.

[A Simple Paint Demo](#)



Pixel Manipulation

Pixel Manipulation

- The "Smudge" tool in the previous demo is implemented by computing the color component values of pixels in the image.
- The implementation requires some way to read the colors of pixels in a canvas. That can be done with the function `graphics.getPixelData(x,y,w,h)`, where `graphics` is a 2D graphics context for the canvas.
- The function reads the colors of a rectangle of pixels, where `(x,y)` is the upper left corner of the rectangle, `w` is its width, and `h` is its height. The parameters are always expressed in pixel coordinates.
- This returns the color data for a 20-by-10 rectangle in the upper left corner of the canvas.

```
colors = graphics.getImageData(0,0,20,10)
```

- The return value, `colors`, is an object with properties `colors.width`, `colors.height`, and `colors.data`. The `width` and `height` give the number of rows and columns of pixels in the returned data.

Pixel Manipulation

```
colors = graphics.getImageData(0,0,20,10)
```

- The value of `colors.data` is an array, with four array elements for each pixel.
- The four elements contain the red, blue, green, and alpha color components of the pixel, given as integers in the range 0 to 255.
- For a pixel that lies outside the canvas, the four component values will all be zero.
- The array is a value of type `Uint8ClampedArray` whose elements are 8-bit unsigned integers limited to the range 0 to 255. This is one of JavaScript's typed array datatypes, which can only hold values of a specific numerical type.
- Suppose that we just want to read the RGB color of one pixel, at coordinates (x, y) .

```
pixel = graphics.getImageData(x,y,1,1);
```

- Then the RGB color components for the pixel are `R = pixel.data[0]`, `G = pixel.data[1]`, and `B = pixel.data[2]`.

Pixel Manipulation – The Smudge Tool

- The function `graphics.putImageData(imageData, x, y)` is used to copy the colors from an image data object into a canvas, placing it into a rectangle in the canvas with upper left corner at `(x, y)`.
- The `imageData` object can be one that was returned by a call to `graphics.getImageData()`, possibly with its color data modified.
- Or we can create a blank image data object by calling `graphics.createImageData(w, h)` and fill it with data.
- In the smudge tool of the previous demo, when the user clicks the mouse with this tool, we use `OSG.getImageData()` to get the color data from a 9-by-9 square of pixels surrounding the mouse location.
- Since we need to perform real-number arithmetic with color values, we should copy the color components into another typed array of type `Float32Array`, which can hold 32-bit floating point numbers.

Pixel Manipulation – The Smudge Tool

- The floating point array, `smudgeColorArray`, will be used for computing new color values for the image as the mouse moves.
- The color values from this array will be copied into the image data object, `smudgeImageData`, which will then be used to put the color values into the image.

```
function grabSmudgeData(x, y) { // (x,y) gives mouse location
    let colors = OSG.getImageData(x-5,y-5,9,9);
    if (smudgeColorArray == null) {
        // Make image data & array the first time this function is called.
        smudgeImageData = OSG.createImageData(9,9);
        smudgeColorArray = new Float32Array(colors.data.length);
    }
    for (let i = 0; i < colors.data.length; i++) {
        // Copy the color component data into the Float32Array.
        smudgeColorArray[i] = colors.data[i];
    }
}
```

Pixel Manipulation – The Smudge Tool

- The floating point array, **smudgeColorArray**, will be used for computing new color values for the image as the mouse moves.
- The color values from this array will be copied into the image data object, **smudgeImageData**, which will then be used to put the color values into the image.

```
function swapSmudgeData(x, y) { // (x,y) is new mouse location
  let colors = OSG.getImageData(x-5,y-5,9,9); // get color data from image
  for (let i = 0; i < smudgeColorArray.length; i += 4) {
    // The color data for one pixel is in the next four array locations.
    if (smudgeColorArray[i+3] && colors.data[i+3]) {
      // alpha-components are non-zero; both pixels are in the canvas;
      // (getImageData() gets 0 for the alpha value at pixel coordinates
      // that are not actually part of the canvas).
      for (let j = i; j < i+3; j++) { // compute new RGB values
        let newSmudge = smudgeColorArray[j]*0.8 + colors.data[j]*0.2;
        let newImage = smudgeColorArray[j]*0.2 + colors.data[j]*0.8;
        smudgeImageData.data[j] = newImage;
        smudgeColorArray[j] = newSmudge;
      }
      smudgeImageData.data[i+3] = 255; // alpha component
    }
    else {
      // one of the alpha components is zero; set the output
      // color to all zeros, "transparent black", which will have
      // no effect on the color of the pixel in the canvas.
      for (let j = i; j <= i+3; j++) {
        smudgeImageData.data[j] = 0;
      }
    }
  }
  OSG.putImageData(smudgeImageData,x-5,y-5); // copy new colors into canvas
}
```

In-Class Demo

For another example of pixel manipulation, we can look at image filters that modify an image by replacing the color of each pixel with a weighted average of the color of that pixel and the 8 pixels that surround it. Depending on the weighting factors that are used, the result can be as simple as a slightly blurred version of the image, or it can be something more interesting.

[Image Filter Demo](#)

The filtering operation in the demo uses the image data functions `getImageData`, `createImageData()`, and `putImageData()` that were discussed above. Color data from the entire image is obtained with a call to `getImageData()`. The results of the averaging computation are placed in a new image data object, and the resulting image data is copied back to the image using `putImageData()`.





DOUGLAS COLLEGE

HTML Canvas and Images

Images on Canvas

- An image on a web page is specified by an element in the web page:

```

```

- The `src` attribute specifies the URL from which the image is loaded.
- The optional `id` can be used to reference the image in JavaScript:

```
image = document.getElementById("mypic");
```

- This is a reference to the object that represents the image in the document structure.
- Once we have such an object, we can use it to draw the image on a canvas. If `graphics` is a graphics context for the canvas, then

```
graphics.drawImage(image, x, y);
```

- Draws the image with its upper left corner at `(x, y)`.
- Both the point `(x, y)` and the `image` itself are transformed by any transformation in effect in the graphics context.

Images on Canvas

- We can also specify the width and height of the rectangle in which the image is drawn:

```
graphics.drawImage(image, x, y, width, height);
```

- In the above example, the image is scaled to fit the specified rectangle.
- What if the image we want to draw onto the canvas is not part of the web page?
- It is possible to load the image dynamically. This is much like making an off-screen canvas, but we will make an "off-screen image."
- Use the document object to create an `img` element:

```
newImage = document.createElement("img");
```

- An `img` element needs a `src` attribute that specifies the URL from which it is to be loaded. For example:

```
newImage.src = "pic2.jpg";
```

-

Once `src` is set, the browser starts loading the image asynchronously, so the computer continues to execute the script without waiting for the load to complete.

Images on Canvas

- If we want to draw the image after it has finished loading, we need to assign a function to the image's **onload** property before setting the **src**.
- That function will be called when the image has been fully loaded.
- Putting this together, here is a simple JavaScript function for loading an image from a specified URL and drawing it on a canvas after it has loaded:

```
function loadAndDraw( imageURL, x, y ) {  
    let image = document.createElement("img");  
    image.onload = doneLoading;  
    image.src = imageURL;  
    function doneLoading() {  
        graphics.drawImage(image, x, y);  
    }  
}
```

- A similar technique is used to load the images in the filter demo.

Images on Canvas

- When discussing the use of an off-screen canvas in the Simple Paint Demo earlier, we noted that the contents of the off-screen canvas have to be copied to the main canvas, but we didn't say how.
- It is done using `drawImage()`.
- In addition to drawing an image onto a canvas, `drawImage()` can be used to draw the contents of one canvas into another canvas.
- In the demo, the command

```
graphics.drawImage( OSC, 0, 0 );
```

- is used to draw the off-screen canvas to the main canvas.
- Here, **graphics** is a graphics context for drawing on the main canvas, and **OSC** is the object that represents the off-screen canvas.

References

- Interactive Computer Graphics, a Top-Down Approach with WebGL, 7th Edition – Edward Angel and Dave Shreiner, Pearson.
- David J. Eck, Introduction to Computer Graphics, Version 1.3, August 2021. Notes from this source are licensed under a [Creative Commons Attribution-Noncommercial-ShareAlike 4.0 License](#).



Thank you.
DOUGLASCOLLEGE

DOUGLASCOLLEGE

CMPT 2361

Computer Graphics

Shahriar Khosravi, Ph.D.

Lecture 4

Plan for this Lecture

- OpenGL 1.1
 - OpenGL Primitives
 - OpenGL Color
 - The Depth Test
- 3D Coordinates and Transforms
 - Translation, Scaling, and Rotation
 - Hierarchical Modelling
- Projection and Viewing
 - (Too) Many Coordinate Systems
 - The Viewport Transformation
 - The Projection Transformation
 - The Modelview Transformation
 - A Camera Abstraction



DOUGLAS COLLEGE

OpenGL Primitives

OpenGL Primitives

- OpenGL can draw a limited set of basic shapes such as points, lines, and triangles.
- There is no built-in support for curves or curved surfaces, they must be approximated by simpler shapes.
- These basic shapes are referred to as primitives.
- A primitive in OpenGL is defined by its vertices.
- A vertex is a point in 3D space defined by its x, y, and z coordinates.
- Here is an example of how to draw a triangle:

```
glBegin(GL_TRIANGLES);  
glVertex2f( -0.7, -0.5 );  
glVertex2f( 0.7, -0.5 );  
glVertex2f( 0, 0.7 );  
glEnd();
```

OpenGL Primitives

- Each vertex of the triangle is specified by a call to the function `glVertex2f()`.
- Vertices must be specified between calls to `glBegin()` and `glEnd()`.
- The parameter to `glBegin()` tells which type of primitive is being drawn.
- The `GL_TRIANGLES` primitive allows us to draw more than one triangle: Just specify three vertices for each triangle to be drawn.
- Note that using `glBegin()` and `glEnd()` is not the preferred way to specify primitives, even in OpenGL 1.1.
- We consider them to be a convenient way to learn about vertices and their properties, but not the way that things are typically drawn in modern graphics APIs.
- **OpenGL functions send commands to the GPU. The drawing will not be done until the commands are transmitted.**
- To ensure that that happens, the function `glFlush()` must be called.

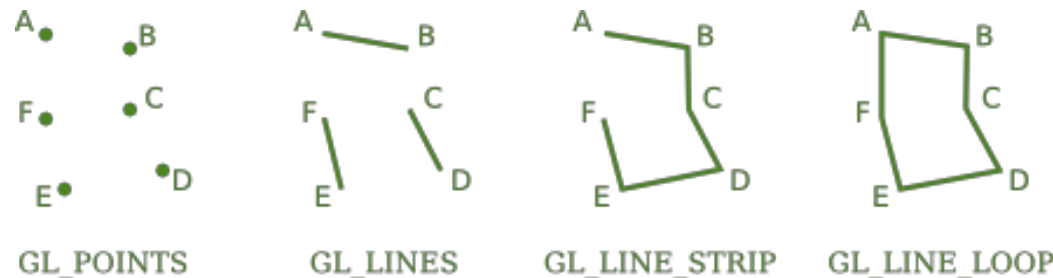


GL_POINTS

- OpenGL 1.1 has ten kinds of primitive, 7 of them still exist in modern OpenGL; the other three have been dropped.
- The simplest primitive is `GL_POINTS`, which simply renders a point at each vertex of the primitive.
- By default, a point is rendered as a single pixel.
- The size of point primitives can be changed by calling `glPointSize(size)`.
- Points are squares by default, circular points can be obtained by calling `glEnable(GL_POINT_SMOOTH)`.
- The functions `glPointSize()` and `glEnable()` change the OpenGL “**state**”. The state includes all the settings that affect rendering.
- The functions `glEnable()` and `glDisable()` can be used to turn many features on and off.
- Any rendering feature that requires extra computation is turned off by default. We have to turn features on by calling `glEnable()` with the appropriate parameter.

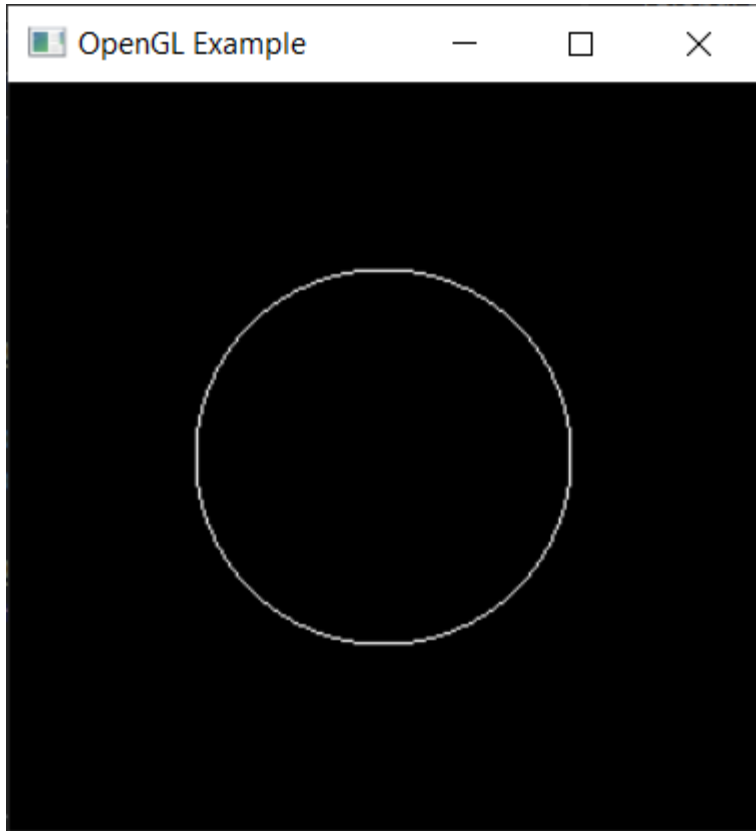
GL_LINES, GL_LINE_STRIP, and GL_LINE_LOOP

- There are three primitives for drawing line segments: `GL_LINES`, `GL_LINE_STRIP`, and `GL_LINE_LOOP`.
- `GL_LINES` draws disconnected line segments, `GL_LINE_STRIP` draws connected line segments, and `GL_LINE_LOOP` draws connected line segments with the last and first vertices connected.



- The points A, B, C, D, E, and F were specified in that order.
- In this illustration, all the points lie in the same plane, but in general, points can be anywhere in 3D space.
- The width for line primitives can be set by calling `glLineWidth(width)`. The line width is always specified in pixels. It is not subject to scaling by transformations.

Example



```
#include <Windows.h>
#include <GL/GL.h>
#include <GL/glut.h>
#include <Math.h> // Needed for sin, cos
#define PI 3.14159265f

void display() {
    glClear(GL_COLOR_BUFFER_BIT); // Clear the color buffer with current clearing color

    double angle, x, y;
    const int SEGMENTS = 64;

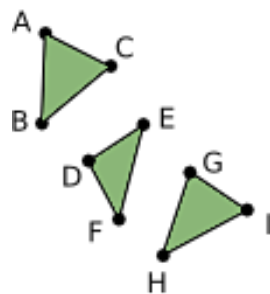
    glBegin(GL_LINE_LOOP);
    for (int i = 0; i < SEGMENTS; i++) {
        angle = 2 * PI * i / SEGMENTS;
        x = 0.5 * cos(angle);
        y = 0.5 * sin(angle);
        glVertex2f(x, y);
    }
    glEnd();

    glFlush(); // Render now
}

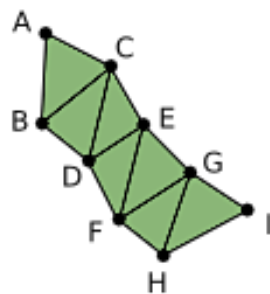
/* Main function: GLUT runs as a console application starting at main() */
int main(int argc, char** argv) {
    glutInit(&argc, argv); // Initialize GLUT
    glutCreateWindow("OpenGL Example"); // Create a window with the given title
    glutInitWindowSize(320, 320); // Set the window's initial width & height
    glutInitWindowPosition(50, 50); // Position the window's initial top-left corner
    glutDisplayFunc(display); // Register display callback handler for window re-paint
    glutMainLoop(); // Enter the event-processing loop
    return 0;
}
```

GL_TRIANGLES

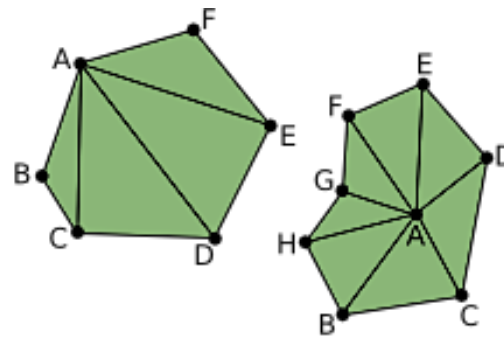
- The next set of primitives is for drawing triangles: `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, and `GL_TRIANGLE_FAN`.
- `GL_TRIANGLES`: every set of three vertices makes a separate triangle.
- `GL_TRIANGLE_STRIP`: the first three vertices produce a triangle. After that, every new vertex adds another triangle to the strip, connecting the new vertex to the two previous vertices.
- `GL_TRIANGLE_FAN`: the first three vertices make a triangle, and every vertex after that adds another triangle, but in this case, the new triangle is made by connecting the new vertex to the previous vertex and to the very first vertex that was specified.
- `GL_TRIANGLE_FAN` can be used for drawing filled-in polygons.



GL_TRIANGLES



GL_TRIANGLE_STRIP

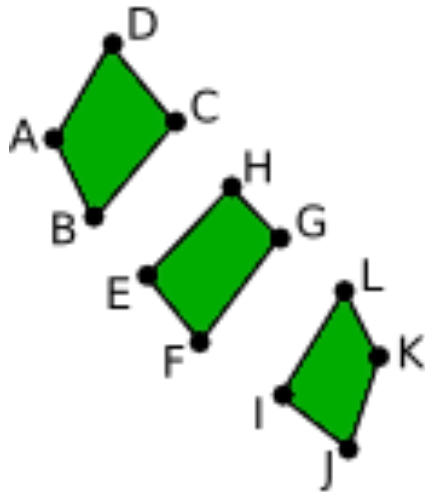


GL_TRIANGLE_FAN

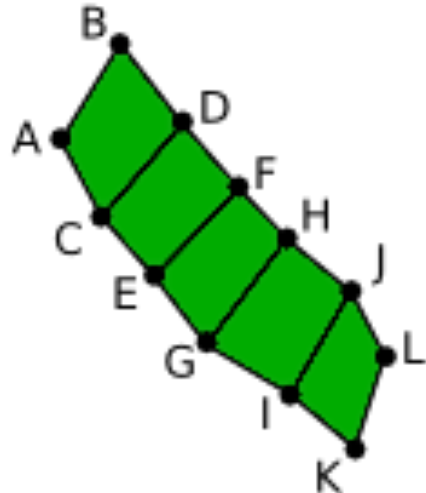
GL_QUADS, GL_QUAD_STRIP, and GL_POLYGON

- These primitives have been removed from modern OpenGL: GL_QUADS, GL_QUAD_STRIP, and GL_POLYGON, but they are still used in OpenGL 1.1.
- The name "quad" is short for quadrilateral, that is, a four-sided polygon.
- In order for a quad to be rendered correctly in OpenGL, all vertices of the quad must lie in the same plane, the same is true for polygon primitives.
- Similarly, to be rendered correctly, **quads and polygons must be convex**.
- Since OpenGL doesn't check whether these conditions are satisfied, **the use of quads and polygons is error-prone**.
- Same shapes can be produced with the triangle primitives, **so these are not necessary**.
- The vertices for these primitives are specified in the order A, B, C, etc.
- For GL_QUADS, the vertices for each individual quad should be specified in **counterclockwise** order around the quad.
- For GL_QUAD_STRIP, the vertices should alternate from one side of the strip to the other.

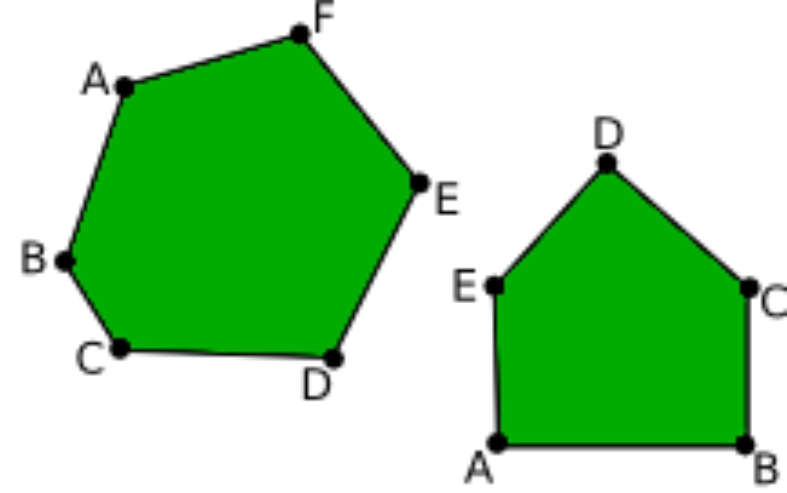
GL_QUADS, GL_QUAD_STRIP, and GL_POLYGON



GL_QUADS



GL_QUAD_STRIP



GL_POLYGON



DOUGLAS COLLEGE

OpenGL Color

OpenGL Color

- OpenGL has a large collection of functions for specifying colors for the geometry that is drawn, called `glColor*()`.
- For realistic 3D graphics, OpenGL has a more complicated notion of color that uses a different set of functions.
- The function `glColor3f()` has three parameters of type float, which give the red, green, and blue components of the color as numbers in the range 0.0 to 1.0.
- Values outside the range 0.0 – 1.0 are allowed and will be used as given in computations, but will be clamped to the range 0 – 1 when appearing on the screen
- The function `glColor4f()` adds the alpha transparency component to the color.
- We need two commands to turn on transparency:

```
glEnable(GL_BLEND);  
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

- Transparency works fine in 2D, it is much more difficult to use transparency correctly in 3D.

OpenGL Color

```
glColor3f(0, 0, 0);           // Draw in black.
glColor3f(1, 1, 1);           // Draw in white.
glColor3f(1, 0, 0);           // Draw in full-intensity red.
glColor3ub(1, 0, 0);          // Draw in a color just a tiny bit different from
                               // black. (The suffix, "ub" or "f", is important!)
glColor3ub(255, 0, 0);        // Draw in full-intensity red.
glColor4f(1, 0, 0, 0.5);      // Draw in transparent red, but only if OpenGL
                               // has been configured to do transparency.
```

OpenGL Color

- Using any of the `glColor*()` functions sets the value of a "current color," which is part of the OpenGL state.
- When we generate a vertex with one of the `glVertex*()` functions, the current color is saved along with the vertex coordinates, as an attribute of the vertex.
- In OpenGL, colors are associated with individual vertices, not with complete shapes.
- By changing the current color between calls to `glBegin()` and `glEnd()`, we can get a shape in which different vertices have different color attributes!
- OpenGL by default will compute the colors of pixels inside the shape by interpolating the colors of the vertices.

```
#include <Windows.h>
#include <GL/GL.h>
#include <GL/glut.h>
```

```
void display() { // Display function will draw the image.
```

```
    glClearColor(0, 0, 0, 1); // (In fact, this is the default.)
    glClear(GL_COLOR_BUFFER_BIT);
```

```
    glBegin(GL_TRIANGLES);
    glColor3f(1, 0, 0); // red
    glVertex2f(-0.8, -0.8);
    glColor3f(0, 1, 0); // green
    glVertex2f(0.8, -0.8);
    glColor3f(0, 0, 1); // blue
    glVertex2f(0, 0.9);
    glEnd();
    // Required to copy color buffer onto the screen.
    glutSwapBuffers();
}
```

```
int main(int argc, char** argv) { // Initialize GLUT and
```

```
    glutInit(&argc, argv);
    // Use single color buffer and no depth buffer.
    glutInitDisplayMode(GLUT_SINGLE);
    // Size of display area, in pixels.
    glutInitWindowSize(500, 500);
    // Location of window in screen coordinates.
    glutInitWindowPosition(100, 100);
    // Parameter is window title.
    glutCreateWindow("GL RGB Triangle");
    // Called when the window needs to be redrawn.
    glutDisplayFunc(display);
    // Run the event loop!
    glutMainLoop();
    return 0;
}
```

OpenGL Color

- A common operation in OpenGL is to clear the drawing area by filling it with a background color.
- The function `glClearColor(r, g, b, a)` sets up a color to be used for clearing the drawing area.
- The parameters are floating point values in the range 0 to 1 and all four color components must be provided.
- The default clear color is black with an alpha component equal to zero.
- The command to do the actual clearing is `glClear( GL_COLOR_BUFFER_BIT )`.
- The correct term for the drawing area is the **color buffer**.
- OpenGL uses several buffers in addition to the color buffer, such as **depth buffer**.
- The `glClear()` command can be used to clear several different buffers at the same time, which can be more efficient than clearing them separately.
- The parameter to `glClear()` tells it which buffer or buffers to clear.

OpenGL Color

- To clear several buffers at once, combine the constants that represent them with an arithmetic OR operation.
- In 3D graphics, the depth buffer plays an essential role, and the appropriate parameter for `glClear()` is `GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT`.
- In 2D graphics, the depth buffer is generally not used, and the appropriate parameter for `glClear()` is just `GL_COLOR_BUFFER_BIT`.

Color and Vertex Arrays

- We have seen that there are versions of `glColor*()` and `glVertex*()` that take different numbers and types of parameters.
- There are also versions that accept a single array parameter for all of the data.
- The names for such versions end with "v".
 - `glColor3fv()`, `glVertex2iv()`, `glColor4ubv()`, and `glVertex3dv()`.
- The "v" actually stands for "vector," meaning a one-dimensional array of numbers.
- For example, in the function call `glVertex3fv(coords)`, `coords` would be an array containing at least three floating point numbers.

```
#include <Windows.h>
#include <GL/GL.h>
#include <GL/glut.h>

void display() { // Display function will draw the image.

    glClearColor(0, 0, 0, 1); // (In fact, this is the default.)
    glClear(GL_COLOR_BUFFER_BIT);

    float coords[] =
        { -0.5, -0.5, 0.5, -0.5, 0.5, 0.5, -0.5, 0.5 };

    glColor3f(0, 0, 1); // blue
    glBegin(GL_TRIANGLE_FAN);
    glVertex2fv(coords); // Uses coords[0] and coords[1].
    glVertex2fv(coords + 2); // Uses coords[2] and coords[3].
    glVertex2fv(coords + 4); // Uses coords[4] and coords[5].
    glVertex2fv(coords + 6); // Uses coords[6] and coords[7].
    glEnd();
    // Required to copy color buffer onto the screen.
    glutSwapBuffers();
}

int main(int argc, char** argv) { // Initialize GLUT and

    glutInit(&argc, argv);
    // Use single color buffer and no depth buffer.
    glutInitDisplayMode(GLUT_SINGLE);
    // Size of display area, in pixels.
    glutInitWindowSize(500, 500);
    // Location of window in screen coordinates.
    glutInitWindowPosition(100, 100);
    // Parameter is window title.
    glutCreateWindow("OpenGL Example");
    // Called when the window needs to be redrawn.
    glutDisplayFunc(display);
    // Run the event loop!
    glutMainLoop();
    return 0;
}
```

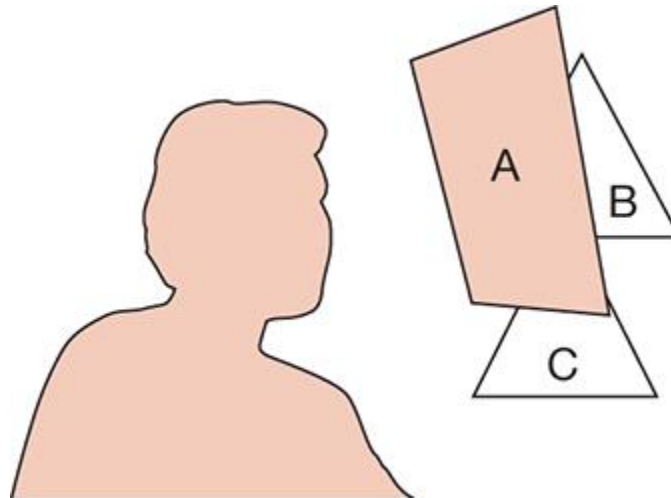


DOUGLAS COLLEGE

The Depth Buffer and Depth Test

The Hidden Surface Problem

- This diagram shows a simplified version of this **hidden-surface problem**.
- From the viewer's position, quadrilateral A is seen clearly, but triangle B is blocked from view, and triangle C is only partially visible.
- Given the position of the viewer and the triangles, we should be able to draw the triangles such that the correct image is obtained.
- Algorithms for ordering objects so that they are drawn correctly are called **visible-surface algorithms** or **hidden-surface-removal** algorithms.
- WebGL has a particular algorithm associated with it, **the Z-buffer algorithm**.

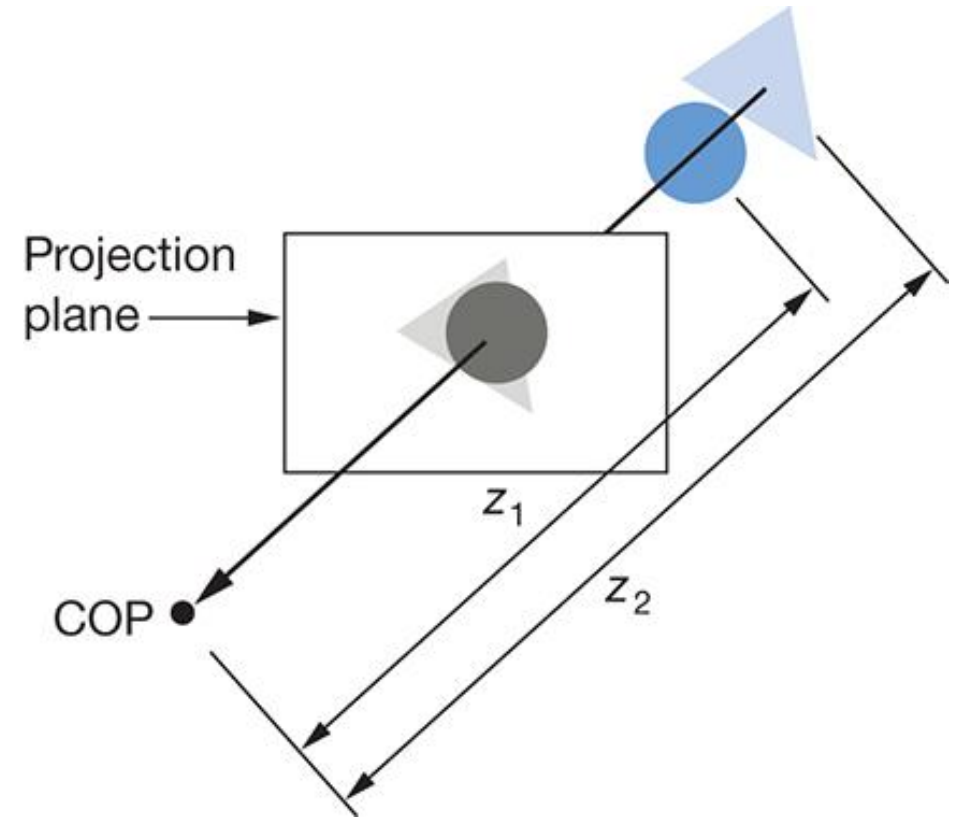


The Hidden Surface Problem

- Hidden-surface-removal algorithms can be divided into two broad classes:
 - **Object-space algorithms**: attempt to order the surfaces of the objects in the scene such that rendering surfaces in a particular order provides the correct image. For example, if we were to render the back-facing surfaces first, we could “paint” over them with the front surfaces and produce the correct image. This class of algorithms does not work well with pipeline architectures in which objects are passed down the pipeline in an arbitrary order. In order to decide on a proper order in which to render the objects, the graphics system must have all the objects available so it can sort them into the desired back-to-front order.
 - **Image-space algorithms**: work as part of the projection process and seek to determine the relationship among object points on each projector. **The z-buffer algorithm** is of the latter type and fits in well with the rendering pipeline in most graphics systems because we can save partial information as each object is rendered.

The Z-Buffer Algorithm

- The basic idea of the z-buffer algorithm is shown here.
- A projector from the center of the eye (COP) passes through two surfaces.
- Since the circle is closer to the viewer than to the triangle, it is the circle's color that determines the color placed in the color buffer at the location corresponding to where the projector pierces the projection plane.
- The difficulty is determining how we can make this idea work regardless of the order in which the triangle and the circle pass through the pipeline.



The Z-Buffer Algorithm

- OpenGL stores a depth value for each pixel in the image. The extra memory that is used to store these depth values makes up the **depth buffer** or the Z-buffer.
- The depth buffer is used to keep track of what is currently visible at each pixel.
- When a second object is drawn at that pixel, the depth buffer is used to decide whether the new object is in front of or behind the currently visible object.
- If the new object is in front, the color of the pixel is changed to show the new object, and the depth buffer is also updated.
- If the new object is behind, the data for the new object is discarded and the color and depth buffers are left unchanged.
- By default, the depth buffer is not turned on. We can enable the depth test by calling:

```
glEnable( GL_DEPTH_TEST );
```

In-Class Demo

This demo lets you experiment with the depth test. It also lets you see what happens when part of your geometry extends outside the visible range of z-values.

[Cube and Depth Test](#)



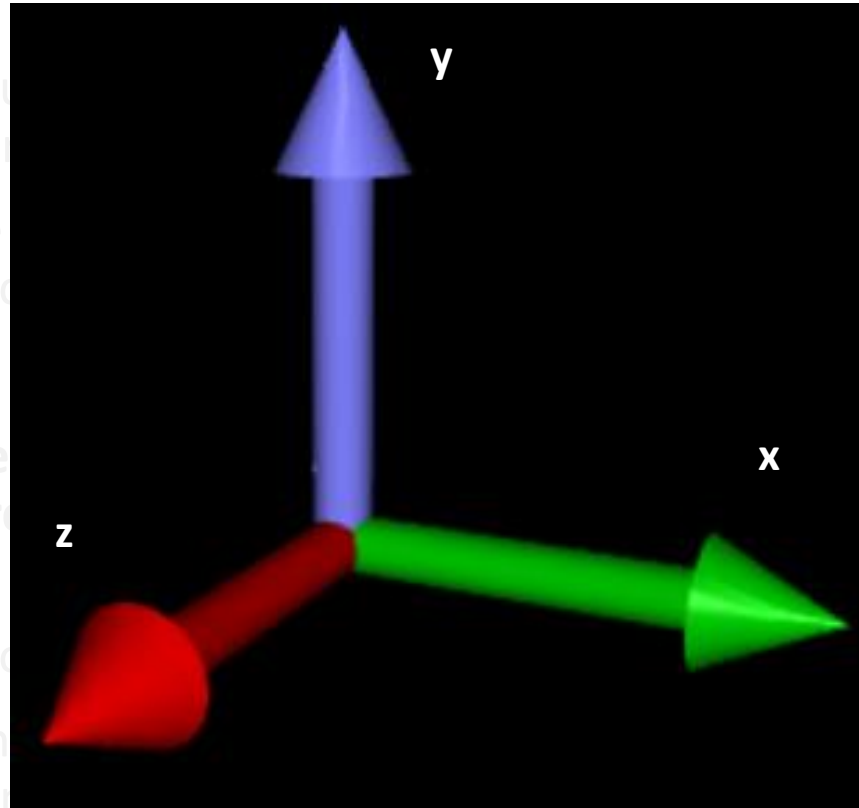
3D Coordinates

3D Coordinates

- In OpenGL, the usual coordinate system used by programmers is one with the x- and y-axes lying in the plane of the screen.
- The z-axis is perpendicular to the screen with the positive direction of the z-axis pointing out of the screen towards the viewer.
- The default coordinate system in OpenGL is similar but has the positive direction of the z-axis pointing into the screen.
- The coordinate system that is actually used is arbitrary and set up by a transformation.
- **The convention in OpenGL is to work with a coordinate system in which the positive z-direction points toward the viewer and the negative z-direction points away from the viewer.**
- The transformation into default coordinates reverses the direction of the z-axis.
- The conventional arrangement of the axes produces a **right-handed coordinate system**. The default OpenGL coordinate system is a **left-handed system**.

3D Coordinates

- In OpenGL, the usual coordinate system is one with the x- and y-axes lying in the plane of the screen.
- The z-axis is perpendicular to the plane of the screen, pointing out of the screen toward the viewer.
- The default coordinate system has the x-axis pointing to the right, the y-axis pointing up, and the z-axis pointing into the screen.
- The coordinate system can be transformed by a transformation.
- **The convention in OpenGL is that the positive z-direction points toward the viewer.**
- The transformation into the normalized device coordinates is done by a transformation.
- The conventional arrangement of the axes is the **right-handed coordinate system**. The default OpenGL coordinate system is the right-handed coordinate system.



3D Transforms

3D Translation, Scaling, and Rotation

- A translation by (dx, dy, dz) transforms a point (x, y, z) to the point $(x+dx, y+dy, z+dz)$.
- In OpenGL, this translation would be specified by the command `glTranslatef( dx, dy, dz );` or by `glTranslated( dx, dy, dz );`
- The OpenGL command for scaling is `glScale*()`, where the "*" can be either "f" or "d".
- For example, `glScalef( sx, sy, sz );` transforms a point (x, y, z) to $(x*sx, y*sy, z*sz)$.
- Rotation in 3D is more involved than in 2D. In 2D, rotation is rotation about a point, which is usually taken to be the origin.
- In 3D, rotation is rotation about a line, which is called the **axis of rotation**.
- Any line can be an axis of rotation, but we generally use an axis that passes through the origin.

In-Class Demo

The following demo helps you understand rotation about an axis in three-dimensional space. Use the buttons labeled "+X", "-X", and so on to make the cube rotate about the coordinate axes, or enter any (x,y,z) point and click "Set". Drag your mouse on the image to rotate the scene.

[3D Rotation Demo](#)



3D Translation, Scaling, and Rotation

- A translation by (dx, dy, dz) transforms a point (x, y, z) to the point $(x+dx, y+dy, z+dz)$.
- In OpenGL, this translation would be specified by the command `glTranslatef(dx, dy, dz);`
- The command `glScalef(sx, sy, sz);` scales an object by factors of "s" or "d".
- For example, `glScalef(2, 2, 2);` scales an object by a factor of 2 in each dimension.
- **Transforms are applied to objects that are drawn after the transformation function is called, and transformations apply to objects in the opposite order of the order in which they appear in the code.**
- Rotation in 3D is more involved than in 2D. In 2D, rotation is rotation about a point, which is usually taken to be the origin.
- In 3D, rotation is rotation about a line, which is called the **axis of rotation**.
- Any line can be an axis of rotation, but we generally use an axis that passes through the origin.

3D Rotation

- There is an easy way to specify a line that passes through the origin: Just specify one other point that is on the line, in addition to the origin.
- In OpenGL, an axis of rotation is specified by three numbers, (ax, ay, az) , which are not all zero.
- The axis of rotation is the line through $(0, 0, 0)$ and (ax, ay, az) .
- To specify a rotation transformation in 3D, we have to specify an axis and the angle of rotation about that axis.
- To define the direction of rotation in 3D, we use the **right-hand rule** (in a right-handed coordinate system):
 - Point the thumb of your right hand in the direction of the axis (from the point $(0,0,0)$ towards the point (ax,ay,az)). The direction of rotation for positive angles is given by the direction in which your fingers curl.

3D Rotation

```
glScalef(2, 2, 2);           // Uniform scaling by a factor of 2.
glScalef(0.5, 1, 1);         // Shrink by half in the x-direction only.

glScalef(-1, 1, 1);          // Reflect through the yz-plane.
                             // Reflects the positive x-axis onto negative x.

glTranslatef(5, 0, 0);        // Move 5 units in the positive x-direction.

glTranslatef(3, 5, -7.5);     // Move each point (x,y,z) to (x+3, y+5, z-7.5).

glRotatef(90, 1, 0, 0);       // Rotate 90 degrees about the x-axis.
                             // Moves the +y axis onto the +z axis
                             //   and the +z axis onto the -y axis.

glRotatef(-90, -1, 0, 0);     // Has the same effect as the previous rotation.

glRotatef(90, 0, 1, 0);       // Rotate 90 degrees about the y-axis.
                             // Moves the +z axis onto the +x axis
                             //   and the +x axis onto the -z axis.

glRotatef(90, 0, 0, 1);       // Rotate 90 degrees about the z-axis.
                             // Moves the +x axis onto the +y axis
                             //   and the +y axis onto the -x axis.

glRotatef(30, 1.5, 2, -3);    // Rotate 30 degrees about the line through
                             //   the points (0,0,0) and (1.5,2,-3).
```

Drawing in 2D with OpenGL

- OpenGL does not have 2D transform functions, but we can just use the 3D versions with appropriate parameters.
- For translation by (dx,dy) in 2D, use `glTranslatef(dx, dy, 0)`. The zero translation in the z direction means that the transform doesn't change the z coordinate.
- For scaling by (sx,sy) in 2D, use `glScalef(sx, sy, 1)`, which scales only in the x and y directions, leaving the z coordinate unchanged.
- For rotation through an angle r about the origin in 2D, use `glRotatef(r, 0, 0, 1)`. This is rotation about the z-axis.
 - Rotation by a positive angle will be in the counterclockwise direction in the xy-plane.
 - This is the same convention that we have used previously for the positive direction of rotation.



DOUGLAS COLLEGE

Hierarchical Modelling

Hierarchical Modelling

- In hierarchical modeling, an object can be defined in its own natural coordinate system, usually using (0,0,0) as a reference point.
- The object can then be scaled, rotated, and translated to place it into world coordinates or into a more complex object. To implement this, we need a way of limiting the effect of a modeling transformation to one object or to part of an object.
- That can be done using a **stack of transforms**.
 - Before drawing an object, **push** a copy of the current transform onto the stack.
 - After drawing the object and its sub-objects, using any necessary temporary transformations, restore the previous transform by **popping** it from the stack.
- OpenGL 1.1 maintains a stack of transforms and provides functions for manipulating that stack.
- Since transforms are represented as matrices, the stack is actually a stack of matrices. In OpenGL, the functions for operating on the stack are named `glPushMatrix()` and `glPopMatrix()`.

glPushMatrix() and glPopMatrix()

- These functions do not take parameters or return a value.
- OpenGL keeps track of a current matrix, which is the composition of all transforms that have been applied.
- Calling a function such as `glScalef()` simply modifies the current matrix.
- When an object is drawn, using the `glVertex*()` functions, the coordinates that were specified for the object are transformed by the current matrix.
- There is another function that affects the current matrix: `glLoadIdentity()`.
- Calling `glLoadIdentity()` sets the current matrix to be the identity transform, which represents no change of coordinates at all and is the usual starting point for a series of transformations.

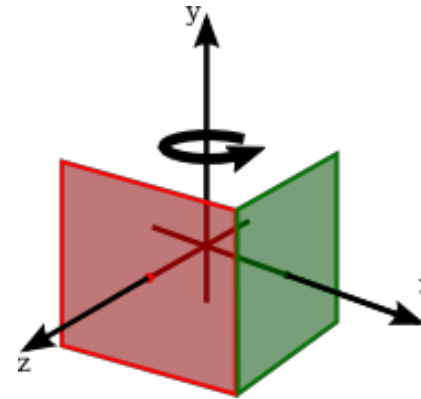
glPushMatrix() and glPopMatrix()

- When `glPushMatrix()` is called, a copy of the current matrix is pushed onto the stack.
- This does **not** change the current matrix; it just saves a copy on the stack.
- When `glPopMatrix()` is called, the matrix on the top of the stack is popped from the stack, and that matrix replaces the current matrix.
- Note that `glPushMatrix()` and `glPopMatrix()` must always occur in corresponding pairs; `glPushMatrix()` saves a copy of the current matrix, and a corresponding call to `glPopMatrix()` restores that copy.
- Between a call to `glPushMatrix()` and the corresponding call to `glPopMatrix()`, there can be additional calls of these functions, as long as they are properly paired.
- Usually, we call `glPushMatrix()` before drawing an object and `glPopMatrix()` after finishing that object.
- In between, drawing sub-objects might require additional pairs of calls to those functions.

glPushMatrix() and glPopMatrix() – example_01.cpp

- Imagine we want to draw a cube by transforming a square multiple times.
- Start with a function that draws a square in the position of the front face of the cube.

```
void square(float r, float g, float b) {
    glColor3f(r, g, b);
    glBegin(GL_TRIANGLE_FAN);
    glVertex3f(-0.5, -0.5, 0.5);
    glVertex3f(0.5, -0.5, 0.5);
    glVertex3f(0.5, 0.5, 0.5);
    glVertex3f(-0.5, 0.5, 0.5);
    glEnd();
}
```



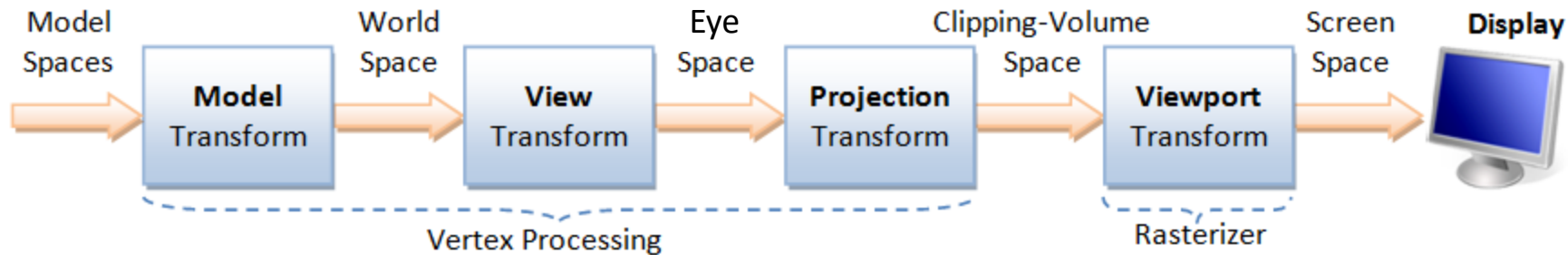
- To make a red front face for the cube, we just need to call `square(1, 0, 0)`.
- To make a right face, we can start with a front face and rotate it 90 degrees about the y-axis.
- The calls to `glPushMatrix()` and `glPopMatrix()` ensure that the rotation that is applied to the square will not carry over to objects that are drawn later.

```
glPushMatrix();
glRotatef(90, 0, 1, 0);
square(0, 1, 0); // green right face
glPopMatrix();
```



Coordinate Systems

Coordinate Systems

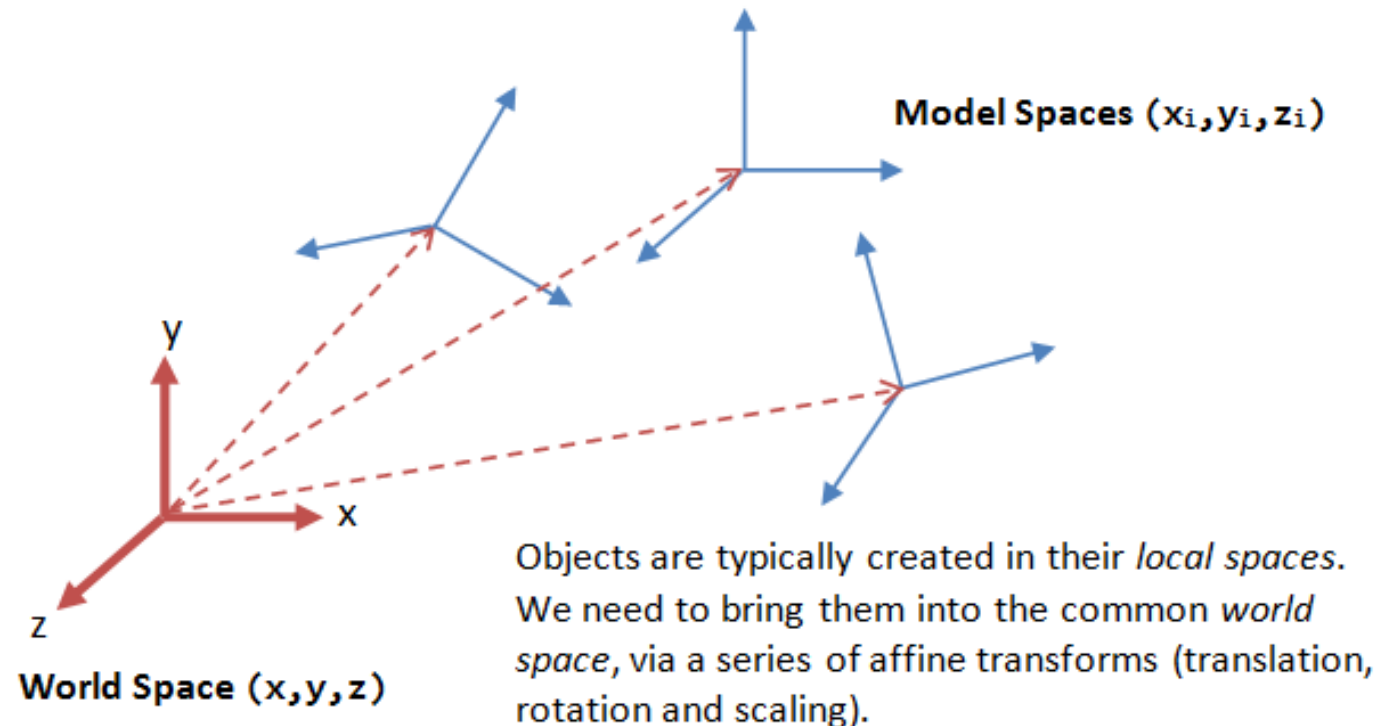


- The process used to produce a 3D scene on the display in Computer Graphics is like taking a photograph with a camera. It involves four transformations:
 1. Arrange the objects in the world (Model Transformation).
 2. Position and orientation the camera (View Transformation).
 3. Select a camera lens, adjust the focus length, and zoom factor to set the camera's field of view (Projection Transformation).
 4. Print the photo on a selected area of the paper (Viewport Transformation).
- OpenGL does **not** keep track of separate modeling and viewing transforms. They are combined into a single transform, which is known as the **modelview transformation**.



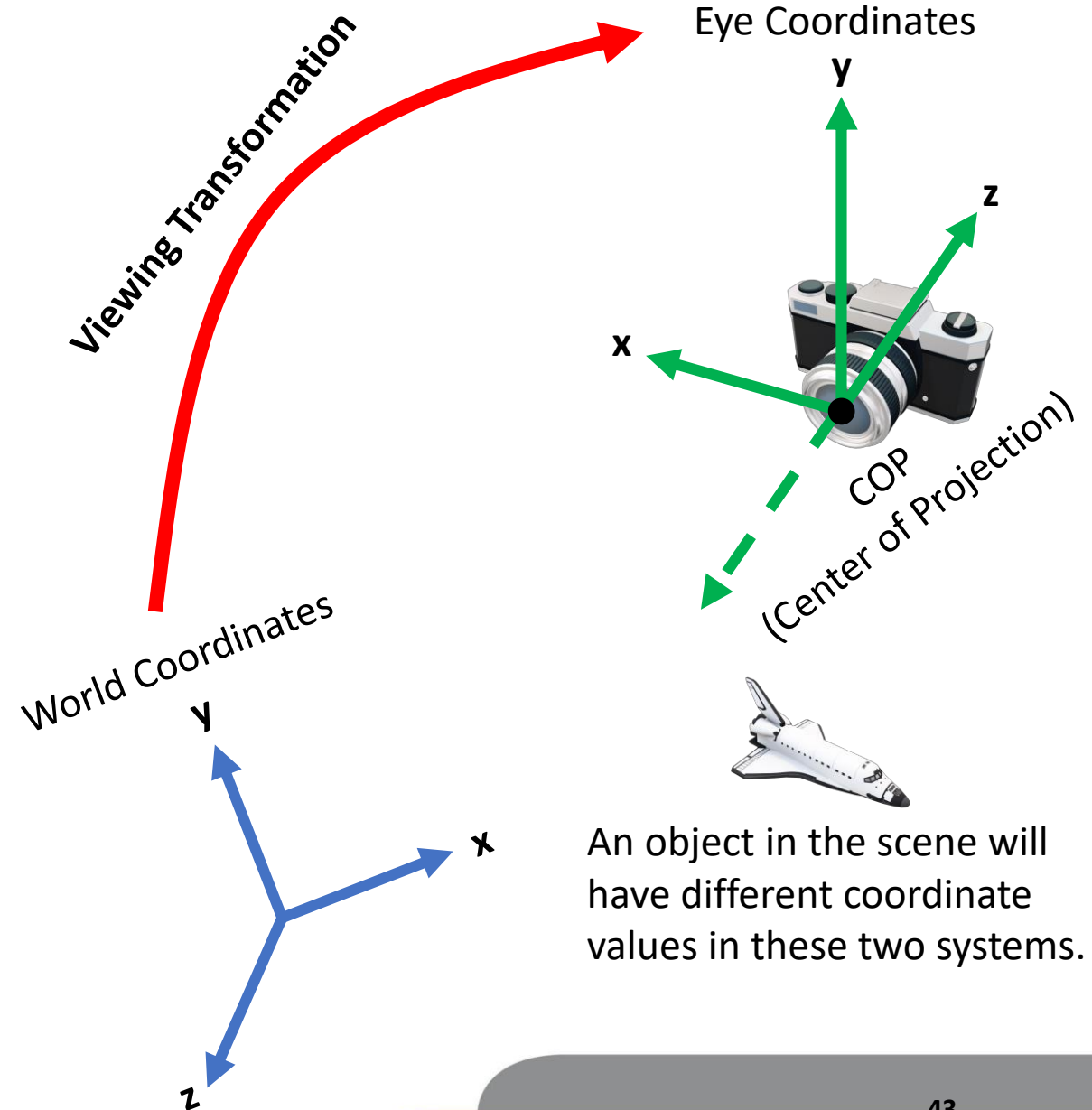
Coordinate Systems

- The coordinates that we use for drawing an object are called **object coordinates**.
- The coordinates in which we build the scene are called **world coordinates**. The **modeling transformation** maps from object coordinates to world coordinates.



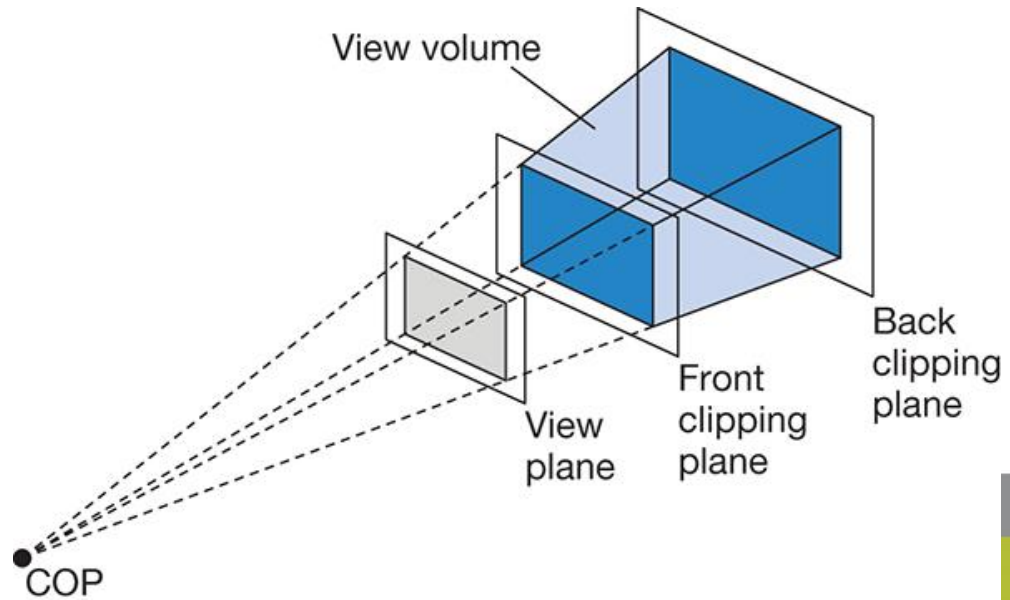
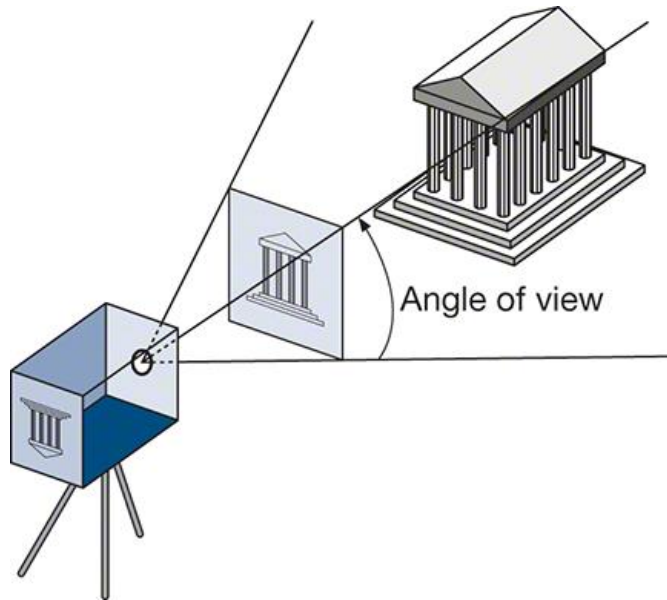
Coordinate Systems

- We cannot make a picture of the scene until we know the position of the "viewer" and where the viewer is looking. In OpenGL, we imagine that the viewer is attached to their own individual coordinate system, which is known as **eye coordinates**.
- In eye coordinates, the viewer is at the origin, $(0,0,0)$, looking in the direction of the negative z-axis; the positive direction of the y-axis is pointing straight up; and the x-axis is pointing to the right.
- The transform from world coordinates to eye coordinates is called the **viewing transformation**.



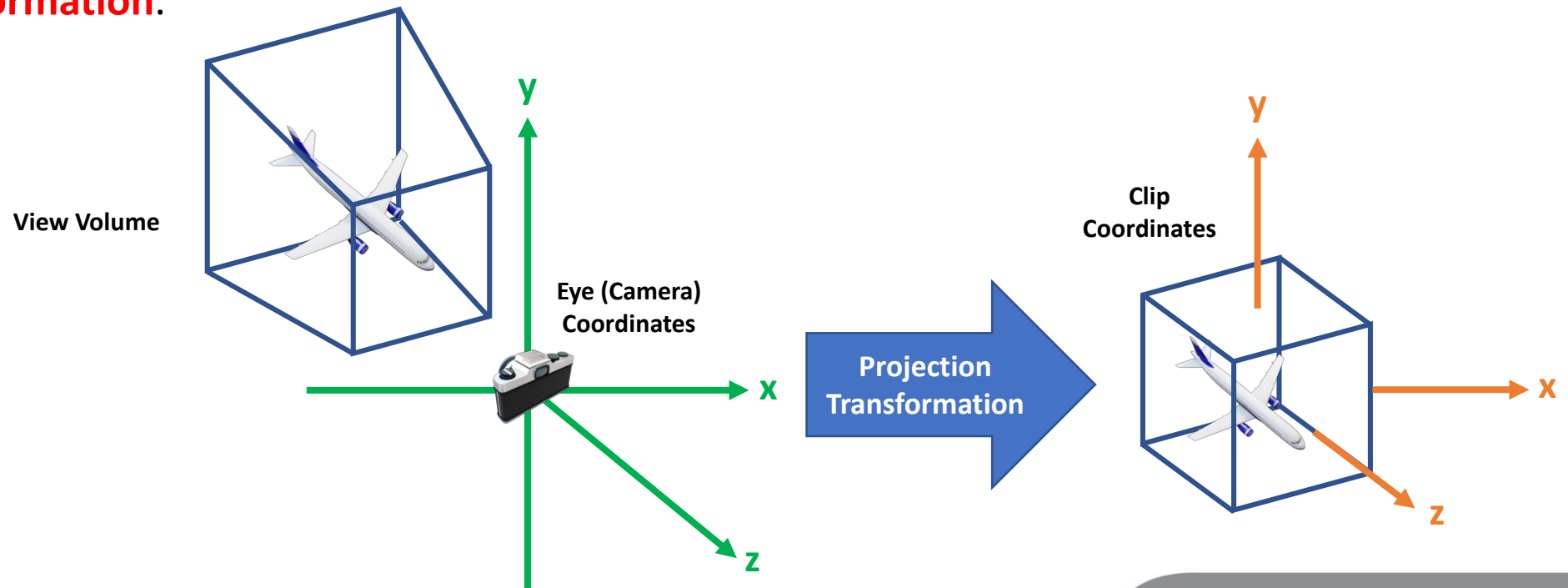
Coordinate Systems

- The viewer cannot see the entire 3D world, only the part that fits into the **viewport**, which is the rectangular region of the screen or other display device where the image will be drawn.
- The scene is "clipped" by the edges of the viewport. Also, in OpenGL, the viewer can see only a limited range of z-values in the eye coordinate system. Points with larger or smaller z-values are clipped away and are not rendered into the image (for the Z-buffer algorithm).
- The volume of space that is actually rendered into the image is called the **view volume**. Things inside the view volume make it into the image. Everything else cannot be seen.



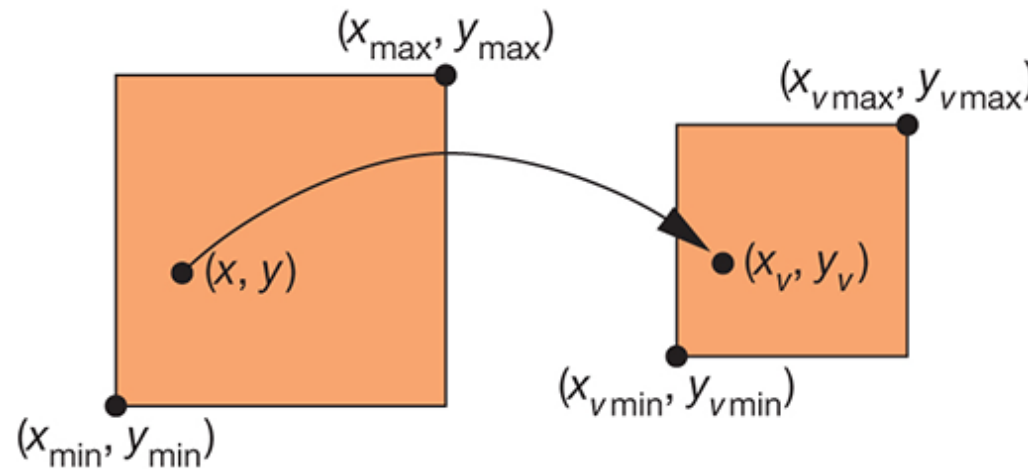
Coordinate Systems

- To draw, OpenGL applies a coordinate transform that maps the view volume onto a cube.
- The cube is centered at the origin and extends from -1 to 1 in the x-direction, in the y-direction, and in the z-direction. The coordinate system on this cube the **clip coordinates**.
- The transformation from eye coordinates to clip coordinates is called the **projection transformation**.



Coordinate Systems

- In the end, when things are actually drawn, there are **device coordinates**, the 2D coordinate system in which the actual drawing takes place on a physical display device.
- Usually, in device coordinates, the pixel is the unit of measure. The drawing region is a rectangle of pixels. This is the rectangle that is called the **viewport**.
- The **viewport transformation** takes x and y from the clip coordinates and scales them to fit the viewport.



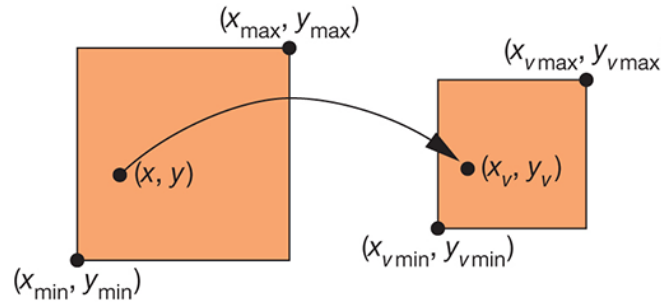


DOUGLAS COLLEGE

The Viewport Transformation

The Viewport Transformation

- The viewport transformation transforms x and y clip coordinates to the coordinates that are used on the display device.



- To specify the viewport transform, it is only necessary to specify the rectangle on the device where the scene will be rendered. This is done using the `glViewport()` function.
- OpenGL must be provided with a drawing surface by the environment in which it is running, i.e., GLUT in our case.
- The drawing surface is a rectangular grid of pixels, with a horizontal size and a vertical size.
- OpenGL uses a coordinate system on the drawing surface that puts $(0,0)$ at the lower left, with y increasing from bottom to top and x increasing from left to right.

The Viewport Transformation

- When the drawing surface is first given to OpenGL, the viewport is set to be the entire drawing surface. However, it is possible for OpenGL to draw to a different rectangle by calling `glViewport( x, y, width, height );`
- Here, (x,y) is the lower left corner of the viewport, in the drawing surface coordinate system, and width and height are the size of the viewport.
- Clip coordinates from -1 to 1 will be mapped to the specified viewport.
- When the size of the drawing surface changes (when the user resizes a window that contains the drawing surface), OpenGL does not automatically update the viewport. However, the environment in which OpenGL is running (GLUT) might do that for us.
- `glViewport()` can be used to draw several different scenes, or several views of the same scene, on the same drawing surface.
- For example, if we want to draw two scenes, side-by-side in a drawing surface that is 600-by-400 pixels:

```
// Draw to left half of the drawing surface.
glViewport(0, 0, 300, 400);
// Draw the first scene.
// ...
// Draw to right half of the drawing surface.
glViewport(300, 0, 300, 400);
// Draw the second scene.
// ...
```



DOUGLAS COLLEGE

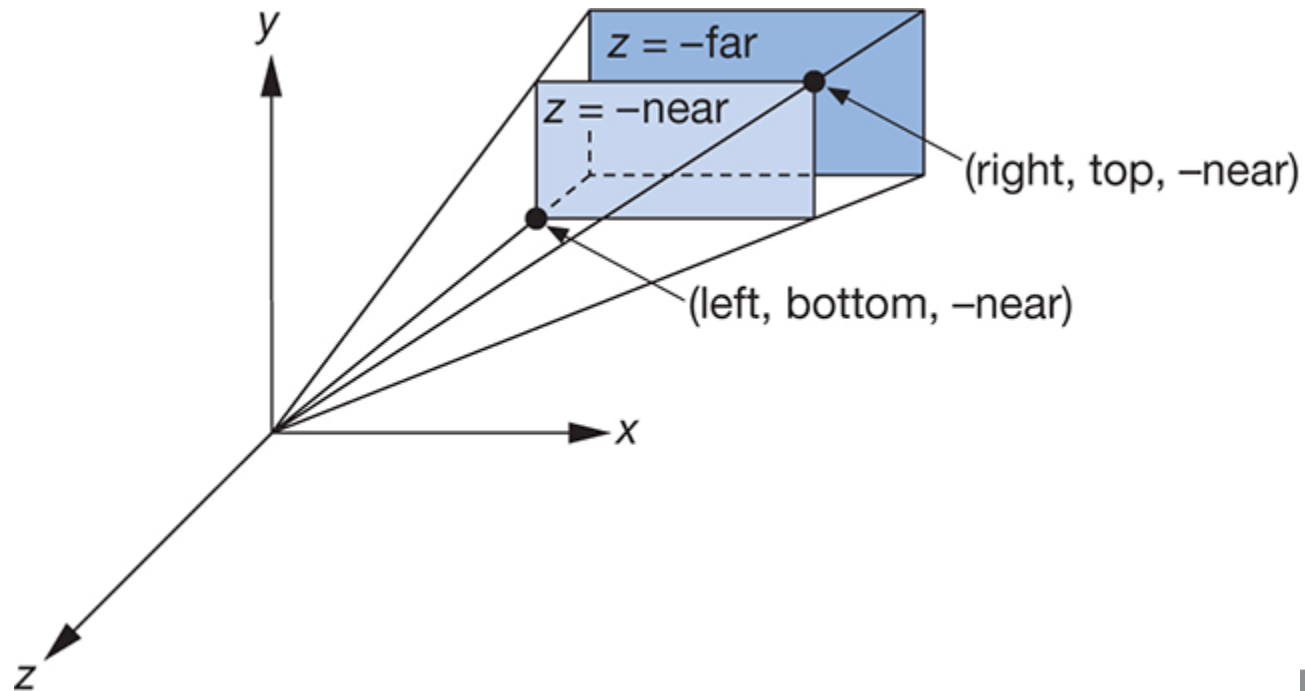
The Projection Transformation

The Projection Transformation

- The projection transformation is represented in OpenGL as a matrix.
- OpenGL keeps track of the projection matrix separately from the matrix that represents the modelview transformation.
- The same transform functions (e.g., `glRotatef()`) can be applied to both matrices, so OpenGL needs some way to know which matrix those functions apply to.
- This is determined by an OpenGL state property called the **matrix mode**. The value of the matrix mode is a constant such as `GL_PROJECTION` or `GL_MODELVIEW`.
- When a function such as `glRotatef()` is called, it modifies a matrix; which matrix is modified depends on the current value of the matrix mode.
- The value is set by calling the function `glMatrixMode()`.
- The initial value is `GL_MODELVIEW`, so if we want to work on the projection matrix, we must first call `glMatrixMode(GL_PROJECTION);`

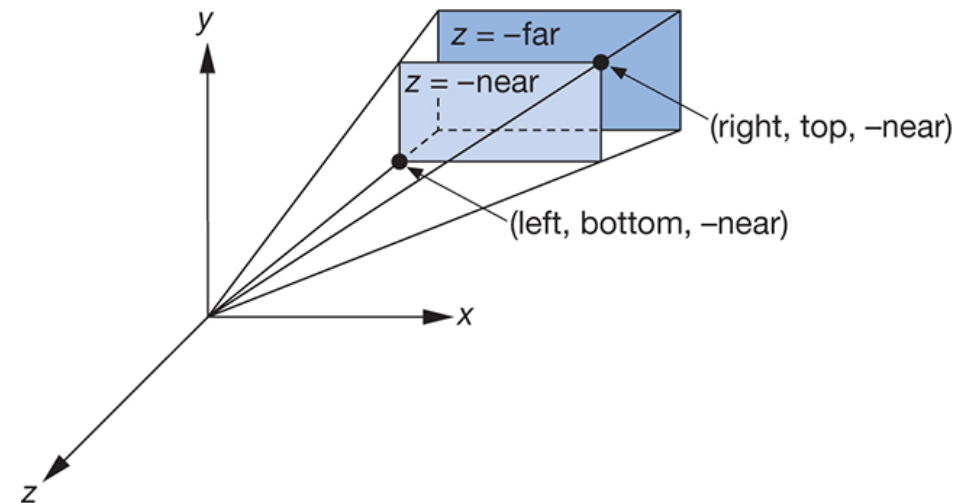
The Perspective Transformation

- There are two general types of projection, **perspective projection** and **orthographic projection**.
- In a **perspective** view, the apparent size of an object depends on how far it is away from the viewer. Only things that are in front of the viewer can be seen.



The Perspective Transformation

- Ignoring clipping in the z-direction, the part of the world that is in view is an infinite pyramid, with the viewer at the apex of the pyramid, and with the sides of the pyramid passing through the sides of the viewport rectangle.
- Since the depth buffer can only store a finite range of depth values, it can't represent the entire range of depth values for the infinite pyramid that is theoretically in view.
- Only objects in a certain range of distances from the viewer can be part of the image. That range of distances is specified by two values, **near** and **far**.
- Anything that is closer to the viewer than the near distance or farther away than the far distance is discarded and does not appear in the rendered image.
- The volume of space that is represented in the image is a "truncated pyramid" or a "frustum".
- This pyramid is the **view volume** for a perspective projection

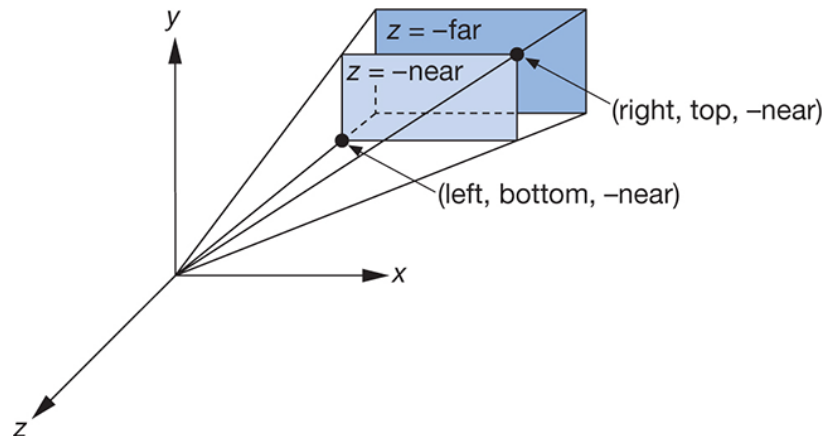


The Perspective Transformation

- The perspective projection transformation maps the six sides of the truncated pyramid in eye coordinates to the six sides of the clipping cube in clip coordinates.
- In OpenGL, setting up the projection transformation is equivalent to defining the view volume.
- For a perspective transformation, we set up a view volume that is a truncated pyramid.
- A perspective transformation can be set up with the `glFrustum()` command:

`glFrustum( xmin, xmax, ymin, ymax, near, far );`

- The first four parameters specify the sides of the pyramid: `xmin`, `xmax`, `ymin`, and `ymax` specify the horizontal and vertical limits of the view volume at **the near clipping plane**.



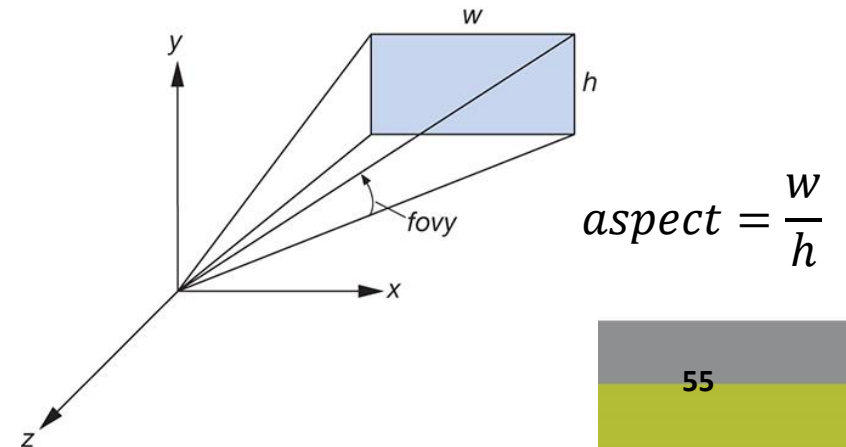
The Perspective Transformation

- `glFrustum()` is often used in a code segment of the form:

```
glMatrixMode(GL_PROJECTION);  
glLoadIdentity();  
glFrustum(xmin, xmax, ymin, ymax, near, far);  
glMatrixMode(GL_MODELVIEW);
```

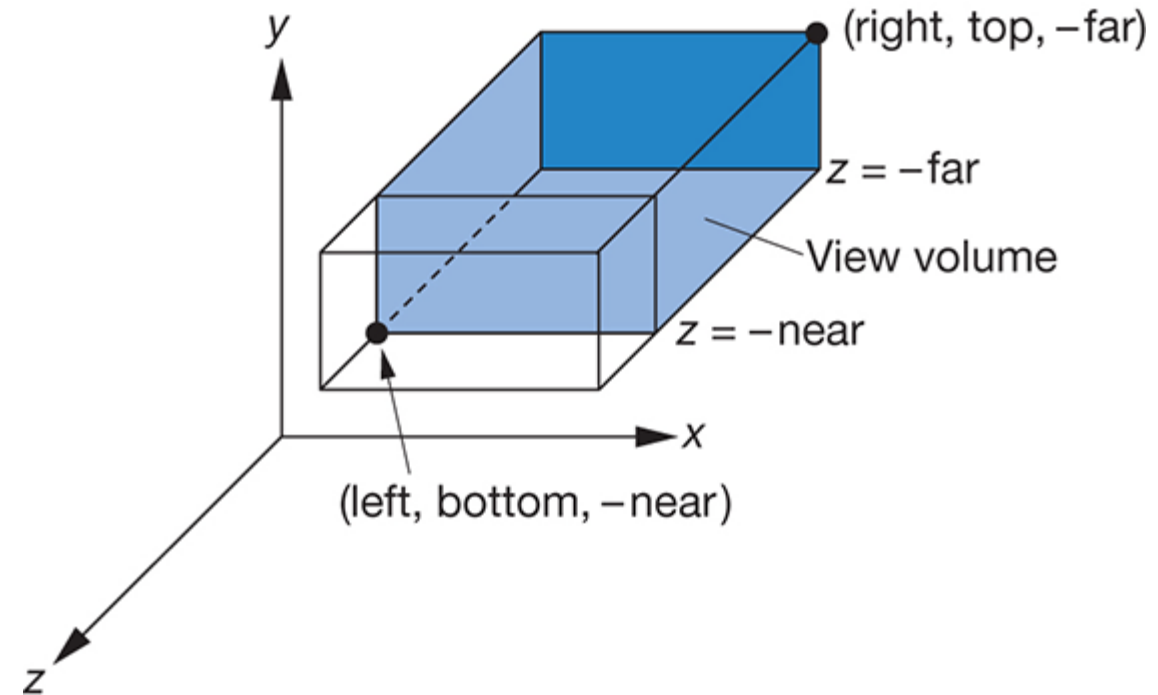
- The call to `glLoadIdentity()` ensures that the starting point is the identity transform.
- This is important since `glFrustum()` modifies the existing projection matrix rather than replacing it.
- There is a library known as GLU that contains some utility functions for use with OpenGL. The GLU library includes the function `gluPerspective()` as an easier way to set up a perspective projection.

```
gluPerspective(fovy, aspect, near, far);
```



The Orthographic Transformation

- In an orthographic projection, the 3D world is projected onto a 2D image by discarding the z -coordinate of the eye-coordinate system.
- This type of projection is unrealistic in that it is not what a viewer would see because the apparent size of an object does not depend on its distance from the viewer.
- Objects in back of the viewer as well as in front of the viewer can be visible in the image.
- Orthographic projections are still useful, especially in interactive modeling programs where it is useful to see true sizes and angles, undistorted by perspective.



The Orthographic Transformation

- An orthographic projection can be set up in OpenGL using the `glOrtho()` function:
`glOrtho( xmin, xmax, ymin, ymax, near, far );`
- The first four parameters specify the x- and y-coordinates of the left, right, bottom, and top of the view volume.
- It is often the case that `near = -far`, and if that is true, then the minimum and maximum z-values will be `near` and `far`.
- Suppose that we want the view volume to be the box centered at the origin containing x, y, and z values in the range from -10 to 10:

```
glMatrixMode(GL_PROJECTION);  
glLoadIdentity();  
glOrtho(-10, 10, -10, 10, -10, 10);  
glMatrixMode(GL_MODELVIEW);
```

The Modelview Transformation

The Modelview Transformation

- OpenGL combines modelling and viewing into a single transformation called the **modelview transformation**.
- This is because there is no way to distinguish between them in principle; the difference is purely conceptual (e.g., translating a model to the left is the same as moving the viewport to the right).
- This even works for scaling: If the viewer shrinks, it will look to the viewer exactly the same as if the world is expanding, and vice-versa.
- It can be difficult to set up a view by combining rotations, scalings, and translations, so OpenGL provides an easier way to set up a typical view. The command is not part of OpenGL itself but is part of the GLU library.

```
gluLookAt( eyeX,eyeY,eyeZ, refX,refY,refZ, upX,upY,upZ );
```

- This method places the viewer at the point (eyeX,eyeY,eyeZ), looking towards the point (refX,refY,refZ). The viewer is oriented so that the vector (upX,upY,upZ) points upwards in the viewer's view.

Putting them all together...

```
// set clear color here, if not set elsewhere
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// set up the projection here, if not done elsewhere
glMatrixMode(GL_MODELVIEW);

glLoadIdentity();

// Viewing transform
gluLookAt(eyeX, eyeY, eyeZ, refX, refY, refZ, upX, upY, upZ);

glPushMatrix();
// apply modeling transform and draw an object
// ...
glPopMatrix();

glPushMatrix();
// apply another modeling transform and draw another object
// ...
glPopMatrix();
```



Thank you.
DOUGLASCOLLEGE

DOUGLASCOLLEGE

CMPT 2361

Computer Graphics

Shahriar Khosravi, Ph.D.

Lecture 5

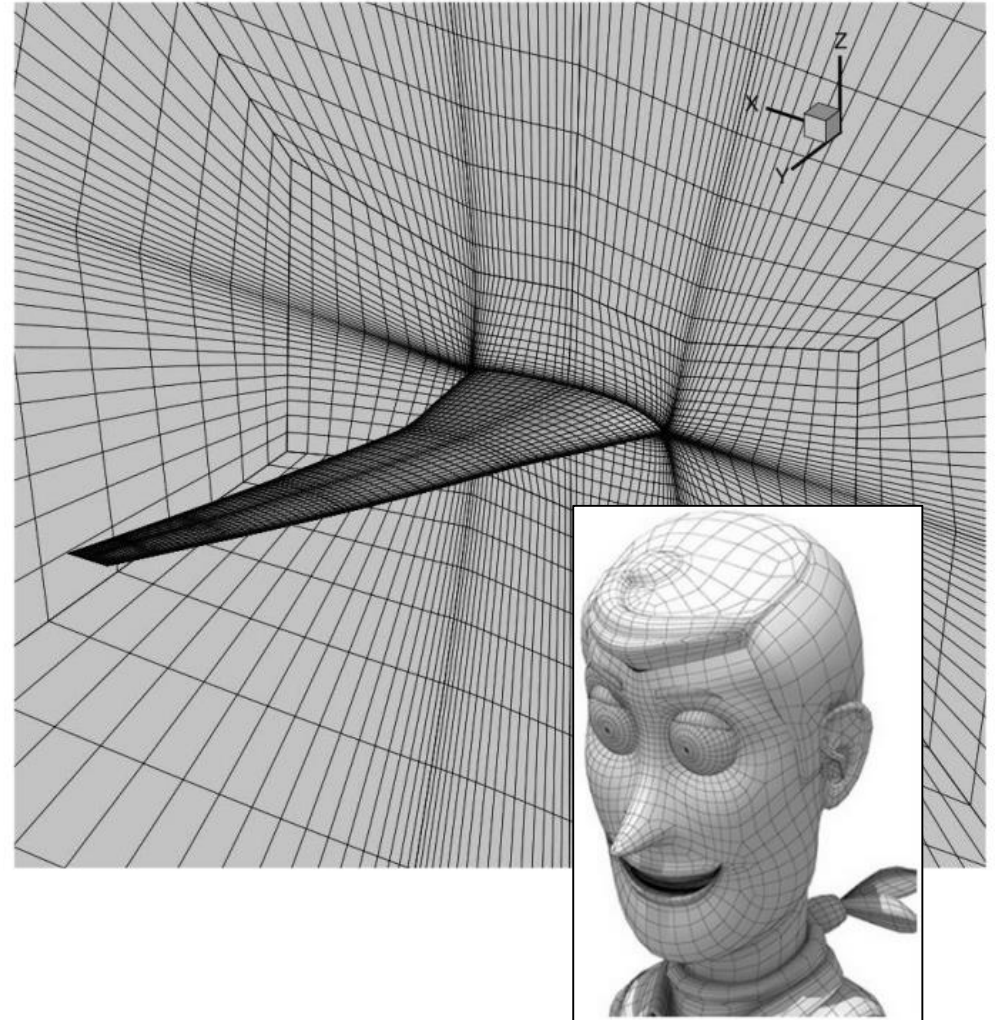
Plan for this Lecture

- Polygonal Meshes and `glDrawArrays()`
 - Indexed Face Sets
 - `glDrawArrays()` and `glDrawElements()`
 - Display Lists and VBOs
- Introduction to Lighting and Material (Phong Model)

Polygonal Meshes

Polygonal Meshes in 3D

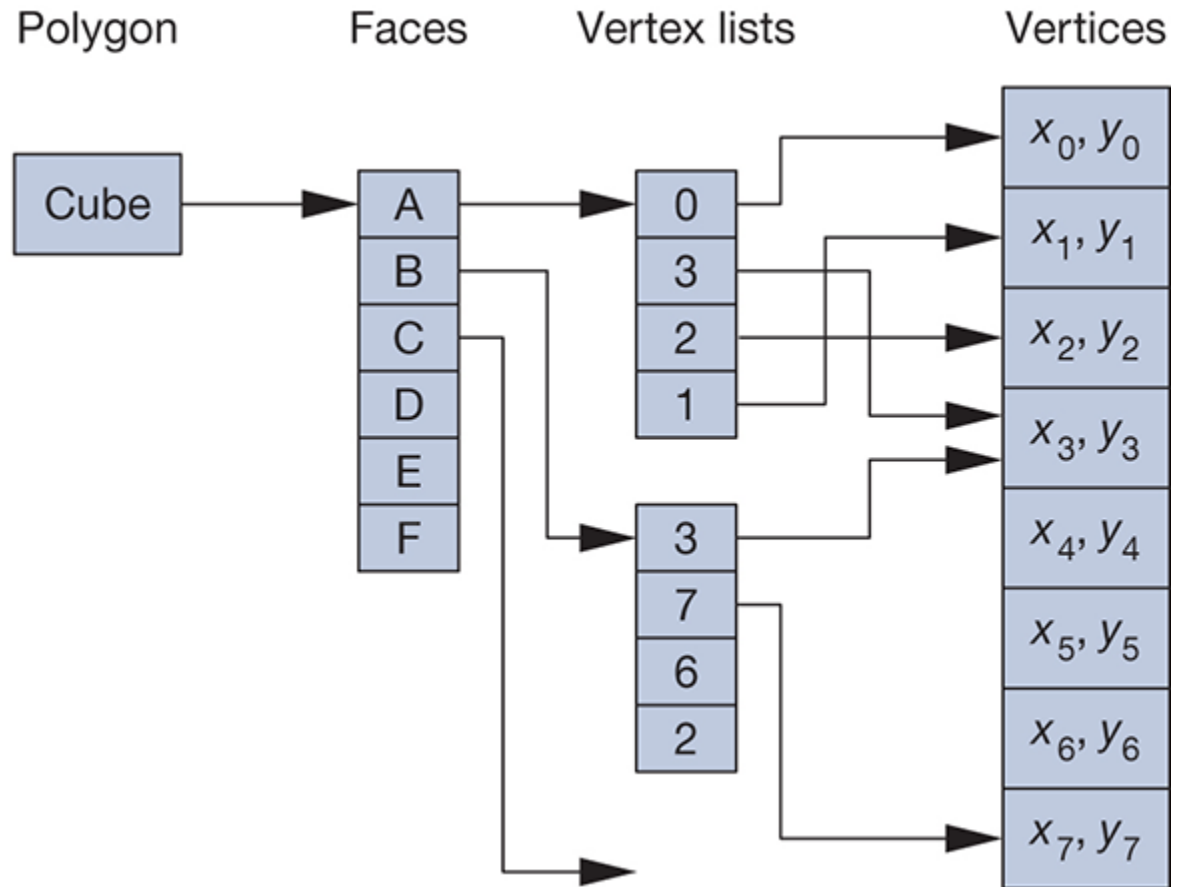
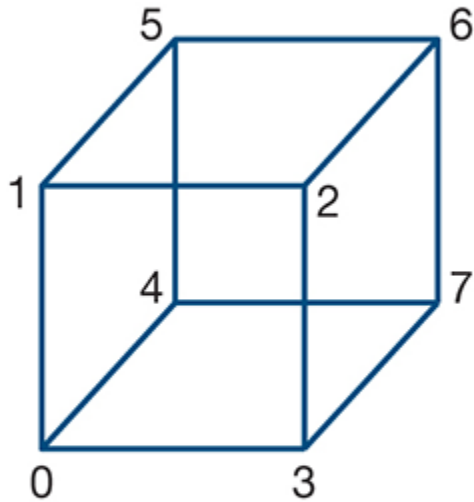
- OpenGL can only directly render points, lines, and polygons.
- A polyhedron, the 3D analog of a polygon, can be represented exactly, since a polyhedron has faces that are polygons.
- A curved surface, such as the surface of a sphere, can only be approximated.
- A polyhedron can be represented, or a curved surface can be approximated, as a polygonal mesh, which is a set of polygons that are connected along their edges.
- If the polygons are small, the approximation can look like a curved surface.





Indexed Face Sets

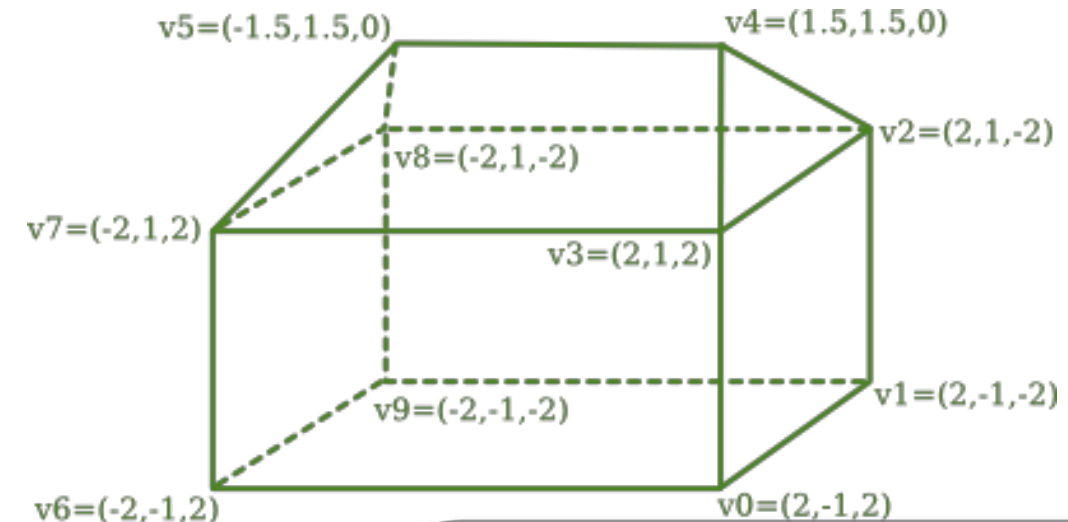
Indexed Face Sets as an Abstract Data Type



Indexed Face Sets

- The polygons in a polygonal mesh are also referred to as "faces" (as in the faces of a polyhedron), and one of the primary means for representing a polygonal mesh is as an **indexed face set**, or **IFS**.
- The data for an IFS includes a list of all the vertices that appear in the mesh, giving the coordinates of each vertex.
- A vertex can then be identified by an integer that specifies its index, or position, in the list.
- As an example, this "house" is a polyhedron with 10 vertices and 9 faces:
- The order in which the vertices are listed is arbitrary.

Vertex #0. (2, -1, 2)
 Vertex #1. (2, -1, -2)
 Vertex #2. (2, 1, -2)
 Vertex #3. (2, 1, 2)
 Vertex #4. (1.5, 1.5, 0)
 Vertex #5. (-1.5, 1.5, 0)
 Vertex #6. (-2, -1, 2)
 Vertex #7. (-2, 1, 2)
 Vertex #8. (-2, 1, -2)
 Vertex #9. (-2, -1, -2)



Indexed Face Sets

- To describe one of the polygonal faces of a mesh, we just have to list its vertices, in order going around the polygon.
- For an IFS, we can specify a vertex by giving its index in the list.
- Here is the face data for the house (numbers in parentheses are indices into the vertex list):
- The order in which the faces are listed is arbitrary.
- A polygon in 3D can be viewed from either side; it has two faces, facing in opposite directions.
- The vertices of a polygon should be listed in **counterclockwise order when looking at the front face of the polygon**.
- **When looking at the back face, the vertices will be listed in clockwise order.**
- This is the default rule used by OpenGL.

Face #0: (0, 1, 2, 3)

Face #1: (3, 2, 4)

Face #2: (7, 3, 4, 5)

Face #3: (2, 8, 5, 4)

Face #4: (5, 8, 7)

Face #5: (0, 3, 7, 6)

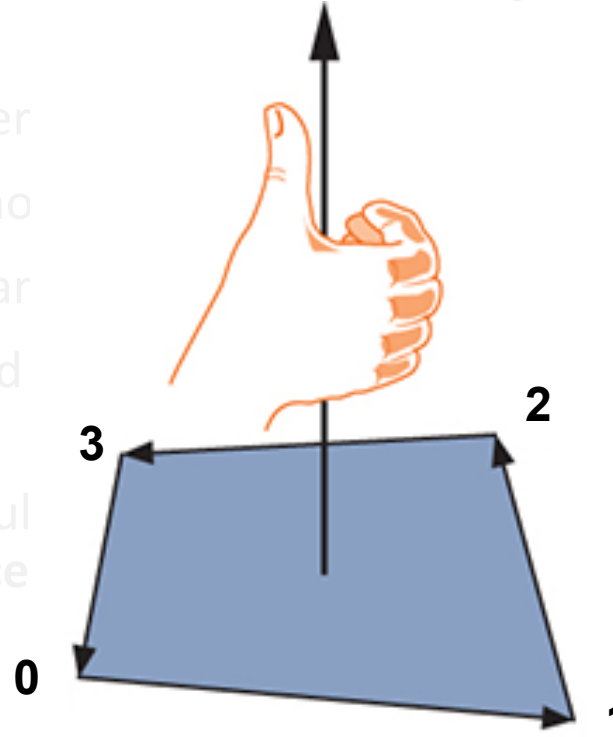
Face #6: (0, 6, 9, 1)

Face #7: (2, 1, 9, 8)

Face #8: (6, 7, 8, 9)

Indexed Face Sets

- To describe one of the polygonal faces of a mesh, we just have to list its vertices, in order going around the polygon.
- For an IFS, we can specify a vertex list.
- Here is the face data for the hot air balloon (the numbers are indices into the vertex list):
- The order in which the faces are listed is important.
- The order in which the vertices of a face are listed is important.
- A polygon in 3D can be viewed facing in opposite directions.
- The vertices of a polygon should be listed in **clockwise order when looking at the front face**.
- When looking at the back face, the vertices should be listed in counter-clockwise order.**
- This is the default rule used by OpenGL.



Face #0: (0, 1, 2, 3)
 Face #1: (3, 2, 4)
 Face #2: (7, 3, 4, 5)
 Face #3: (2, 8, 5, 4)
 Face #4: (5, 8, 7)
 Face #5: (0, 3, 7, 6)
 Face #6: (0, 6, 9, 1)
 Face #7: (2, 1, 9, 8)
 Face #8: (6, 7, 8, 9)

IFS Representation in C/C++

- In C/C++, we might use one dimensional arrays for the data.
- We can store all the vertex coordinates in a single array.
- The length of the vertex array would be three times the number of vertices, and the data for vertex number N will begin at index $3*N$ in the array.
- For the face list, we have to deal with the fact that not all faces have the same number of vertices.
- A common solution is to add a -1 to the array after the data for each face.
- Since in C/C++, it is not possible to determine the length of a primitive array, we also need variables to store the number of vertices and the number of faces.
- See example_02.cpp for how this can be done for our house example.

IFS Representation in C/C++

```

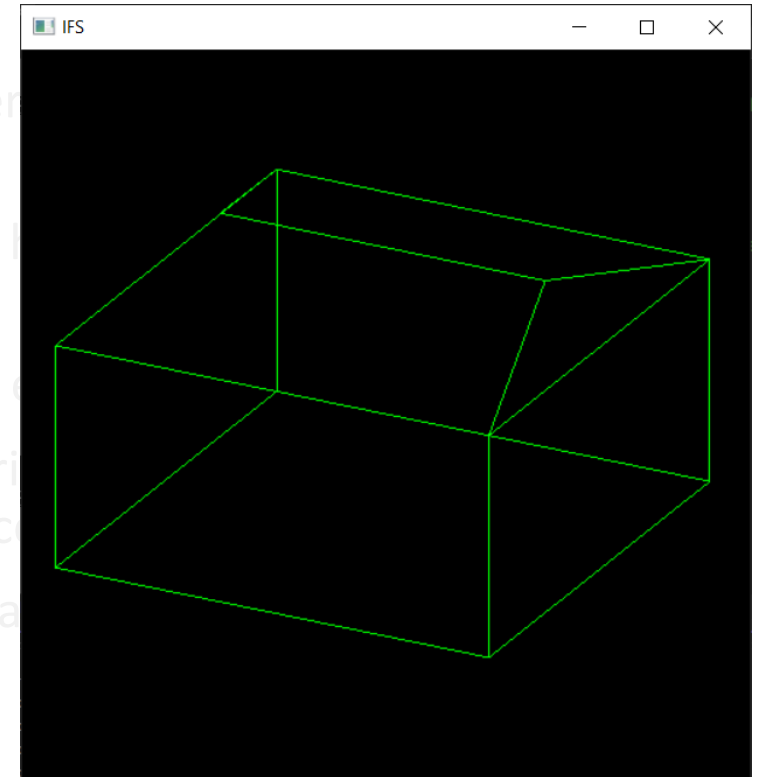
int vertexCount = 10; // Number of vertices.
double vertexData[] =
{ 2,-1,2, 2,-1,-2, 2,1,-2, 2,1,2, 1.5,1.5,0,
  -1.5,1.5,0, -2,-1,2, -2,1,2, -2,1,-2, -2,-1,-2 };

int faceCount = 9; // Number of faces.
int faceData[] =
{ 0,1,2,3,-1, 3,2,4,-1, 7,3,4,5,-1, 2,8,5,4,-1, 5,8,7,-1,
  0,3,7,6,-1, 0,6,9,1,-1, 2,1,9,8,-1, 6,7,8,9,-1 };

int i, j;
j = 0; // index into the faceData array

for (i = 0; i < faceCount; i++) {
    glColor3f(0, 1, 0);
    glBegin(GL_LINE_STRIP);
    while (faceData[j] != -1) { // Generate vertices for face number i.
        int vertexNum = faceData[j]; // Vertex number in vertexData array.
        glVertex3dv(&vertexData[vertexNum * 3]);
        j++;
    }
    j++; // increment j past the -1 that ended the data for this face.
    glEnd();
}

```



glDrawArrays() and glDrawElements()

glDrawArrays() and glDrawElements()

- One complaint about the original OpenGL 1.0 was the large number of function calls needed to draw a primitive using functions such as `glVertex2d()` and `glColor3fv()` with `glBegin()` and `glEnd()`.
- To address this issue, OpenGL 1.1 introduced the functions `glDrawArrays()` and `glDrawElements()`. These functions are still used in modern OpenGL, including WebGL.
- To use `glDrawArrays()`, we send all of the data that is needed to draw a primitive, including vertex coordinates, colors, and other vertex attributes, via arrays.
- Once the data is passed, the primitive can be drawn with a single call to `glDrawArrays()`.
- Since a primitive such as a `GL_LINE_LOOP` or a `GL_TRIANGLES` can include a large number of vertices, the reduction in the number of function calls can be substantial.
- To use `glDrawArrays()`, we must store all of the vertex coordinates for a primitive in a single one-dimensional array of `int`, `float` or `double`.
- Each vertex can have 2, 3, or 4 coordinates.

glDrawArrays()

- Tell OpenGL where to find the data by calling:

```
glVertexPointer(int size, int type, int stride, void* array)
```

number of coordinates
per vertex.

a constant for the data type of
each of the numbers in the array:
GL_FLOAT, GL_INT, and GL_DOUBLE

the distance in bytes between the
location of the data for one vertex
and location for the next vertex.

a pointer that can point to
any type of data. This is the
array that contains the data.

- Enable use of the array by calling:

```
glEnableClientState(GL_VERTEX_ARRAY);
```

- To actually draw the primitive:

```
glDrawArrays( int primitiveType, int firstVertex, int vertexCount)
```

which primitive type
is being drawn.

number of the first vertex
that is to be used for
drawing the primitive.
often equal to 0.

number of vertices to
be used, the total
number of vertices in
the array.

glDrawArrays() – example

- The parameter **firstVertex** is the number of the first vertex that is to be used for drawing the primitive.
- The position is given in terms of vertex number; the corresponding array index is the vertex number times the number of coordinates per vertex, which was set in the call to **glVertexPointer()**.
- The **vertexCount** parameter is the number of vertices to be used.
- Often, **firstVertex** will be zero, and **vertexCount** will be the total number of vertices in the array.

```
#include <Windows.h>
#include <GL/GL.h>
#include <GL/glut.h>
```

```
void display() { // Display function will draw the image.
```

```
    float coords[8] = { -0.5,-0.5, 0.5,-0.5, 0.5,0.5, -0.5,0.5 };
    glVertexPointer(2, GL_FLOAT, 0, coords);
    // we also have to enable use of the array by calling
    glEnableClientState(GL_VERTEX_ARRAY);
    // this corresponds to one use of glBegin/glEnd.
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
    glFlush();
```

```
}
```

```
int main(int argc, char** argv) { // Initialize GLUT and
```

```
    glutInit(&argc, argv);
    // Use single color buffer and no depth buffer.
    glutInitDisplayMode(GLUT_SINGLE);
    // Size of display area, in pixels.
    glutInitWindowSize(500, 500);
    // Location of window in screen coordinates.
    glutInitWindowPosition(100, 100);
    // Parameter is window title.
    glutCreateWindow("glDrawArrays()");
    // Called when the window needs to be redrawn.
    glutDisplayFunc(display);
    // Run the event loop!
    glutMainLoop();
    return 0;
```

```
}
```

glDrawArrays()

- If we want to specify a different color for each vertex, the colors for the vertices can be put into another array. The location of the data is specified by:

```
glColorPointer(int size, int type, int stride, void* array)
```

- This works just like `glVertexPointer()`.
- We also need to enable the color array by calling:

```
glEnableClientState(GL_COLOR_ARRAY);
```

- See `example_01.cpp`, which draws the standard OpenGL red/green/blue triangle.

glDrawElements()

- `glDrawElements()` is similar to `glDrawArrays()`, but it is designed for use with data in a format similar to an indexed face set.
- With `glDrawArrays()`, OpenGL pulls data from the enabled arrays in order, vertex 0, then vertex 1, then vertex 2, and so on.
- With `glDrawElements()`, we provide a list of vertex numbers.
- OpenGL will go through the list of vertex numbers, pulling data for the specified vertices from the arrays.
- As was the case for IFS, the advantage is that the same vertex can be reused several times.
- To use `glDrawElements()`, we need an array to store the vertex numbers. The numbers in the array can be 8, 16, or 32 bit integers.
- We also need arrays to store the vertex coordinates and other vertex data. We must enable those arrays using `glEnableClientState()`.

glDrawElements()

- To actually draw the primitive, call:

`glDrawElements( int primitiveType, vertexCount, dataType, void *array)`

one of the ten primitive
types such as `GL_LINES`.

the number of
vertices to be drawn.

the type of data
in the array, one of
`GL_UNSIGNED_BYTE`,
`GL_UNSIGNED_SHORT`,
or `GL_UNSIGNED_INT`.

- See `example_02.cpp` that uses this function code to draw cubes. Use the arrow keys to rotate the cubes.



DOUGLAS COLLEGE

Display Lists and VBOs

Display Lists and VBOs

- When the same object is going to be drawn more than once, the OpenGL commands and data for drawing that object must be transmitted to the GPU each time the object is drawn.
- This is inefficient.
- We would like to store information on the GPU, so that it can be reused without retransmitting it.
- Two techniques to accomplish this are **display lists** and **vertex buffer objects (VBOs)**.
- Display lists were part of the original OpenGL 1.0, but they are not part of the modern OpenGL API.
- VBOs were introduced in OpenGL 1.5 and are still important in modern OpenGL.

Display Lists

- A **display list** is a list of graphics commands and the data used by those commands.
- Display lists are useful when the same sequence of OpenGL commands will be used several times.
- A display list can be stored in a GPU. The contents of the display list only have to be transmitted once to the GPU. Once a list has been created, it can be "called."
- The key point is that calling a list requires only one OpenGL command.
- Although the same list of commands still has to be executed, only one command has to be transmitted from the CPU to the graphics card, and then the full power of hardware acceleration can be used to execute the commands at the highest possible speed.
- Calling a display list twice can result in two different effects, since the effect can depend on the OpenGL state at the time the display list is called.
- For example, a display list that generates the geometry for a sphere can draw spheres in different locations, as long as different modeling transforms are in effect each time the list is called.

Display Lists

- To use a display list, we first ask for an integer that will identify that list to the GPU:

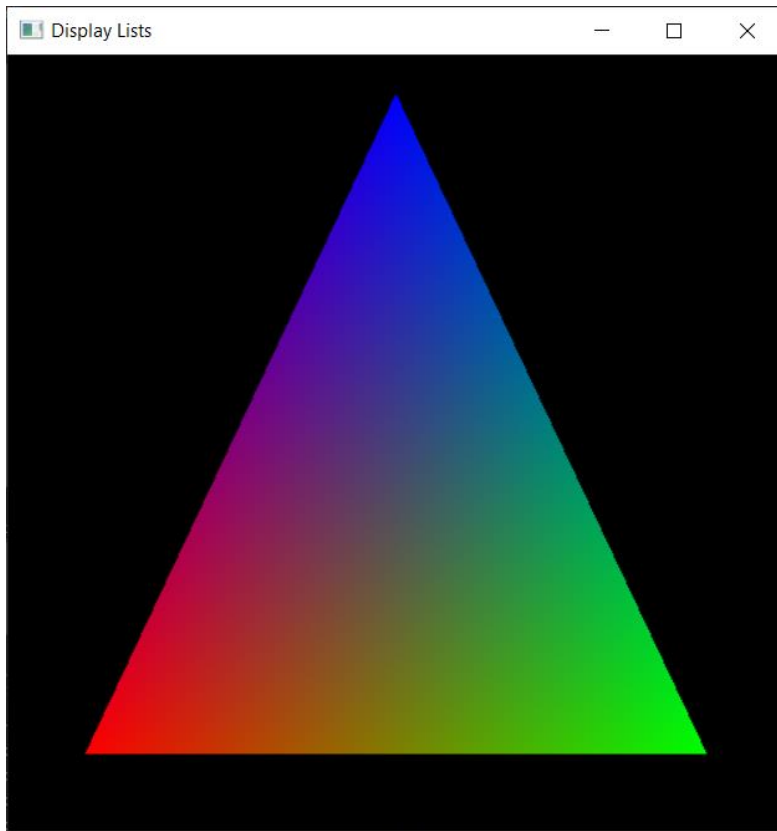
```
int index = glGenLists(1);
```

- The return value is an int which will be the identifier for the list. The parameter to `glGenLists()` is also an `int`, which is usually 1.
- Once a list is allocated, we can store commands into it.

```
glNewList(index, GL_COMPILE);  
// Commands to store  
glEndList();
```

- `GL_COMPILE` means that we only want to store commands into the list, not execute them.
- If we use `GL_COMPILE_AND_EXECUTE`, then the commands will be executed immediately as well as stored in the list for later reuse.
- Call the list with the command: `glCallList(index);`
- If the list is no longer used, delete it using: `glDeleteLists(index, 1);`

Display Lists – Example



```
#include <Windows.h>
#include <GL/GL.h>
#include <GL/glut.h>

void display() { // Display function will draw the image.

    // create one display list
    int index = glGenLists(1);

    // compile the display list, store a triangle in it
    glNewList(index, GL_COMPILE);
    glBegin(GL_TRIANGLES);
    glColor3f(1, 0, 0); // red
    glVertex2f(-0.8, -0.8);
    glColor3f(0, 1, 0); // green
    glVertex2f(0.8, -0.8);
    glColor3f(0, 0, 1); // blue
    glVertex2f(0, 0.9);
    glEnd();
    glEndList();

    // draw the display list
    glCallList(index);
    // delete it if it is not used any more
    glDeleteLists(index, 1);
    glFlush();
}

int main(int argc, char** argv) { // Initialize GLUT and

    glutInit(&argc, argv);
    // Use single color buffer and no depth buffer.
    glutInitDisplayMode(GLUT_SINGLE);
    // Size of display area, in pixels.
    glutInitWindowSize(500, 500);
    // Location of window in screen coordinates.
    glutInitWindowPosition(100, 100);
    // Parameter is window title.
    glutCreateWindow("Display Lists");
    // Called when the window needs to be redrawn.
    glutDisplayFunc(display);
    // Run the event loop!
    glutMainLoop();
    return 0;
}
```

Vertex Buffer Objects (VBOs)

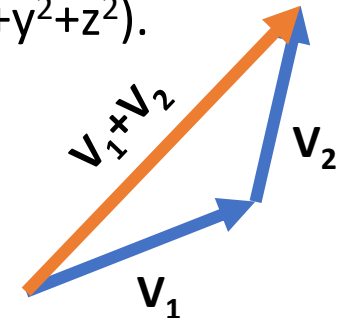
- A VBO is similar to an array in that it stores only data not commands. A VBO is an array that can be stored on the GPU for efficiency of reuse.
- There are OpenGL commands to create and delete VBOs and to transfer data from an array on the CPU side into a VBO on the GPU.
- We can configure `glDrawArrays()` and `glDrawElements()` to take the data from a VBO instead of from an ordinary array in C.
- VBOs were not part of OpenGL 1.1, so we will discuss them later in the course in more details.



Vectors Review

Vectors

- A **vector** is a quantity that has a length and a direction.
- Vectors are often used in computer graphics to represent directions, such as the direction from an object to a light source or the direction in which a surface faces.
- In terms of its coordinates, the length of a 3D vector (x,y,z) is given by $\text{sqrt}(x^2+y^2+z^2)$.
- Vectors of length 1 are called **unit vectors**.
- Dividing a vector by its length is said to **normalize** the vector.
- Two vectors can be added together, and the sum has a geometric meaning:



$$V_1 + V_2 = (x_1+x_2, y_1+y_2, z_1+z_2);$$

- If $V = (x,y,z)$ is a vector and a is a number, then the **scalar product** of a and V is defined as:

$$aV = (a*x, a*y, a*z);$$

- Given two vectors $V_1 = (x_1,y_1,z_1)$ and $V_2 = (x_2,y_2,z_2)$, the **dot product** of V_1 and V_2 is denoted by $V_1 \cdot V_2$ and is defined as:

$$V_1 \cdot V_2 = x_1*x_2 + y_1*y_2 + z_1*z_2$$

Vectors

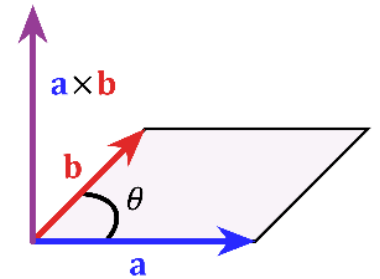
- The dot product is a scalar, **not** a vector.
- The length of a vector V is $\sqrt{V \cdot V}$.
- Furthermore, the dot product of two non-zero vectors V_1 and V_2 has the property that:
- Here, θ is the angle between V_1 and V_2 .
$$\cos(\theta) = \frac{V_1 \cdot V_2}{|V_1| * |V_2|}$$
- In the case of two unit vectors, whose lengths are 1, the dot product of two unit vectors is simply the cosine of the angle between them.
- Since the cosine of a 90-degree angle is zero, two non-zero vectors are perpendicular if and only if their dot product is zero.
- The dot product is particularly important in lighting calculations, where the effect of light shining on a surface depends on the angle that the light makes with the surface.

Vectors

- For vectors $V_1 = (x_1, y_1, z_1)$ and $V_2 = (x_2, y_2, z_2)$, the **cross product** of V_1 and V_2 is denoted $V_1 \times V_2$ and is the vector defined by:

$$V_1 \times V_2 = (y_1 z_2 - z_1 y_2, z_1 x_2 - x_1 z_2, x_1 y_2 - y_1 x_2)$$

- If V_1 and V_2 are non-zero vectors, then $V_1 \times V_2$ is zero if and only if V_1 and V_2 point in the same direction or in exactly opposite directions.
- Assuming $V_1 \times V_2$ is non-zero, then it is perpendicular both to V_1 and to V_2 .
- The vectors V_1 , V_2 , $V_1 \times V_2$ follow the right-hand rule.





Matrices Review

Matrices

- A matrix is a two-dimensional array of numbers.
- A matrix with M rows and N columns is said to be an M-by-N matrix.
- If A and B are matrices, and if the number of columns in A is equal to the number of rows in B, then A and B can be multiplied to give the matrix product AB.
- If A is an N-by-M matrix and B is an M-by-K matrix, then AB is an N-by-K matrix.
- An N-dimensional vector can be thought of an N-by-1 matrix.
- If A is an N-by-N matrix and v is a vector in N dimensions, then the product Av is again an N-dimensional vector.
- The product of a 3-by-3 matrix A and a 3D vector $v = (x,y,z)$ is displayed as:

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_1x + a_2y + a_3z \\ b_1x + b_2y + b_3z \\ c_1x + c_2y + c_3z \end{bmatrix}$$

- **A matrix defines a transformation that can be applied to one vector to yield another vector.**

Matrices

- A **linear transformation** L has the properties that for two vectors v and w , $L(v + w) = L(v) + L(w)$, and for a number s , $L(sv) = sL(v)$.
- Rotation and scaling are linear transformations, **but translation is not a linear transformation**.
- To include translations, we have to widen our view of transformation to include **affine transformations**.
- An affine transformation is a transformation that preserves parallel lines.
- For computer graphics, we are interested in affine transformations in three dimensions. However, we can narrow our view back to linear transformations by moving into the fourth dimension:

$$\begin{bmatrix} a_1 & a_2 & a_3 & t_1 \\ b_1 & b_2 & b_3 & t_2 \\ c_1 & c_2 & c_3 & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- This represents a linear transformation plus a translation in 3D: 

$$\begin{aligned} x_2 &= a_1x + a_2y + a_3z + t_1 \\ y_2 &= b_1x + b_2y + b_3z + t_2 \\ z_2 &= c_1x + c_2y + c_3z + t_3 \end{aligned}$$

Matrices

- In OpenGL, an affine transformation is represented as a 4-by-4 matrix in which the bottom row is (0,0,0,1), and a three-dimensional vector is changed into a four dimensional vector by adding a 1 as the final coordinate.
- The result is that all the affine transformations that are so important in computer graphics can be implemented as multiplication of vectors by matrices.
- The identity transformation, which leaves vectors unchanged, corresponds to multiplication by the identity matrix.
- The OpenGL function `glLoadIdentity()` sets the current matrix to be the 4-by-4 identity matrix.
- An OpenGL transformation function (e.g., `glTranslatef(tx, ty, tz)`) will multiply the current matrix by the 4-by-4 matrix that represents the transformation.
- If M is the current matrix and T is the matrix that represents the transformation, then the current matrix will be set to the product matrix $M \times T$.

Matrices

- For example, the following illustrations show the identity matrix and the matrices corresponding to various OpenGL transformation functions.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Identity Matrix

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

glTranslatef(tx, ty, tz)

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

glScalef(sx, sy, sz)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(d) & -\sin(d) & 0 \\ 0 & \sin(d) & \cos(d) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

glRotatef(d, 1, 0, 0)

$$\begin{bmatrix} \cos(d) & 0 & \sin(d) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(d) & 0 & \cos(d) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

glRotatef(d, 0, 1, 0)

$$\begin{bmatrix} \cos(d) & -\sin(d) & 0 & 0 \\ \sin(d) & \cos(d) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

glRotatef(d, 0, 0, 1)

Matrices

- It is possible to use an arbitrary transformation matrix in OpenGL, using the function `glMultMatrixf(T)` or `glMultMatrixd(T)`.
- The parameter `T` is an array of numbers of type float or double, representing a transformation matrix.
- The array is a one-dimensional array of length 16. The items in the array are the numbers from the transformation matrix, stored in column-major order.
- Column-major order means that the numbers are ordered in the first column, followed by the numbers in the second column, and so on and so forth.
- This is different from what we are used to in C++, for example, when we work with 2D arrays.
- We could use these functions to implement a shear transform, which is not easy to represent as a sequence of scales, rotations, and translations...

Homogenous Coordinates

- There is one common transformation in computer graphics that is **not** an affine transformation: in a perspective projection, the projection transformation is **not** affine.
- In a perspective projection, an object will appear to get smaller as it moves farther away from the viewer, and that is a property that no affine transformation can express.
- we can still represent a perspective projection as a 4-by-4 matrix!
- We have already represented 3D vectors by 4D vectors in which the fourth coordinate is 1. Let us allow the fourth coordinate to be anything, but require that at least one of the four coordinates is non-zero.
- When the fourth coordinate w is non-zero, we can consider the coordinates (x,y,z,w) to represent the three-dimensional vector $(\frac{x}{w}, \frac{y}{w}, \frac{z}{w})$.
- When the fourth coordinate is zero, there is no corresponding 3D vector, but we can think of $(x,y,z,0)$ as representing a 3D point at infinity in the direction of (x,y,z) .
- Coordinates (x,y,z,w) used in this way are known as **homogeneous coordinates**.

Homogenous Coordinates

- If we use homogeneous coordinates, then any 4-by-4 matrix can be used to transform three-dimensional vectors, including matrices whose bottom row is not (0,0,0,1).
- The projection transformation for a perspective projection can now be represented this way in OpenGL.
- In fact, OpenGL represents all three-dimensional points and vectors using homogeneous coordinates, and it represents all transformations as 4-by-4 matrices.
- We can even specify vertices using homogeneous coordinates. For example, the command:
$$\text{glVertex4f}(x, y, z, w);$$
- with a non-zero value for w , generates the 3D point $(x/w, y/w, z/w)$.
- We almost never have to deal with homogeneous coordinates directly.
- The only exception to this is that homogeneous coordinates are used when configuring OpenGL lighting.

Lighting and Material

Lighting and Material

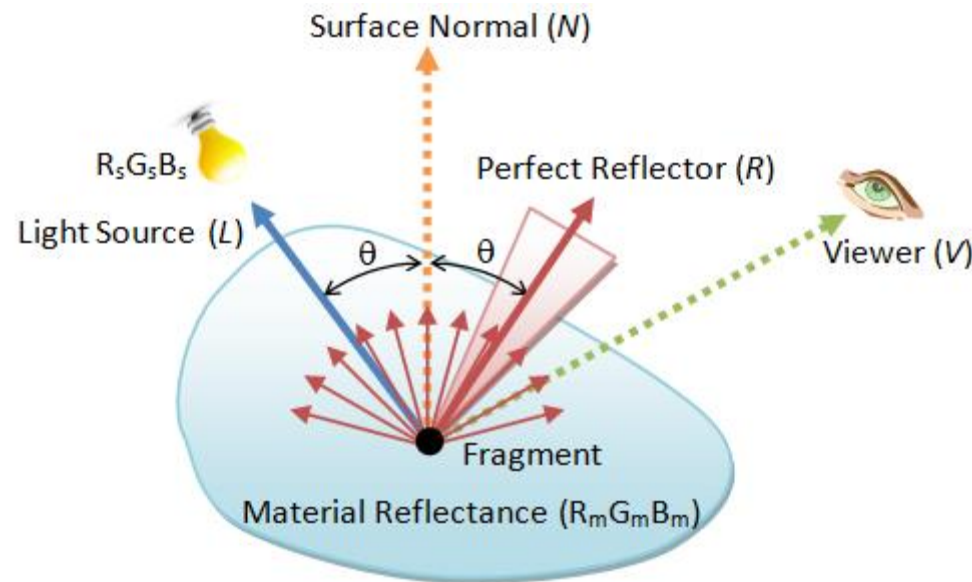
- **Lighting** refers to the handling of interactions between the light sources and the objects in the 3D scene. Lighting is one of the most important factors in producing a realistic scene.
- The color that we see in the real world is the result of the interaction between the light sources and the color material surfaces.
- Three entities are involved: the viewer, light sources, and material.
- When light from a **light source** strikes a surface, some gets absorbed, some is reflected or scattered.
- The angle of reflection depends on the **angle of incidence** and the **surface normal**.
- The amount of scattering depends on the smoothness and the material of the surface.
- The reflected light also spans a certain color spectrum, which depends on the color spectrum of the incident light and the absorption property of the **material**.
- The strength of the reflected light depends on the position and distance of the light source and the viewer, as well as the material.

Lighting and Material

- The reflected light may strike other surfaces, and some is absorbed and some is reflected again.
- The color that we perceive from a surface is the reflected light hitting our eye.
- In a 2D photograph or painting, objects appear to be three-dimensional due to some small variations in colors, known as **shades**.
- To simulate lighting in graphics, we need a lighting model.
- There are two classes of lighting models:
 - **Local illumination**: consider only the direct lightings. The color of the surface depends on the reflectance properties of the surface and the direct lightings.
 - **Global illumination**: in real world, objects receive indirect lighting reflected from other objects and the environment. The global illumination model considers indirect lightings reflected from other objects in the scene. Global illumination model is complex and computationally intensive.

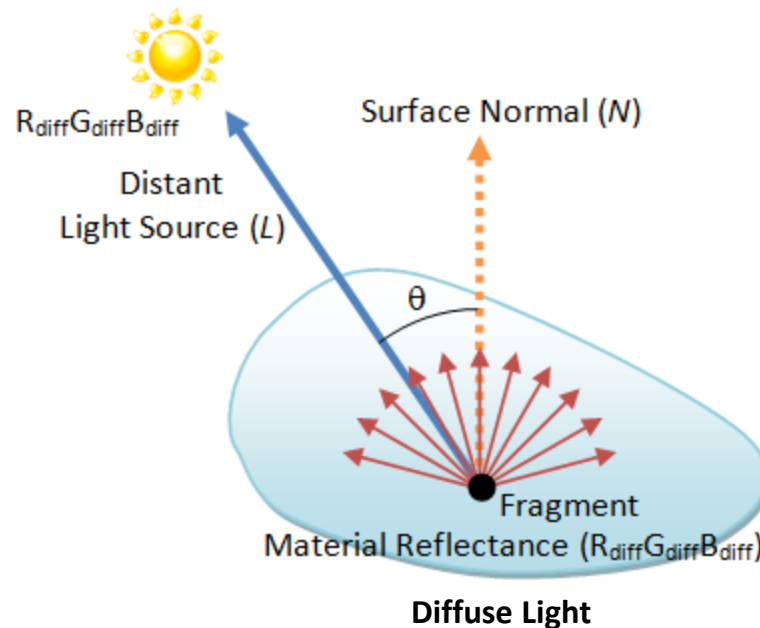
Phong Lighting Model for Light-Material Interaction

- **Phong lighting model** is a local illumination model, which is computationally cheap and extensively used.
- It considers four types of lightings: **diffuse**, **specular**, **ambient**, and **emissive**.
- Consider a fragment P on a surface, four vectors are used: the light source L , the viewer V , the fragment-normal N , and the perfect reflector R .
- The perfect reflector R can be computed from the surface normal N and the incidence light L , according to Newton's law, which states that the angle of incidence is equals to the angle of reflection.



Diffuse Light

- Diffuse light models distant directional light sources (such as the sun).
- The reflected light is scattered equally in all directions, and appears the same to all viewers regardless of their positions, (i.e., independent of viewer vector V).
- The strength of incident light depends on the angle between the light source L and the normal N . This is the dot product between L and N .

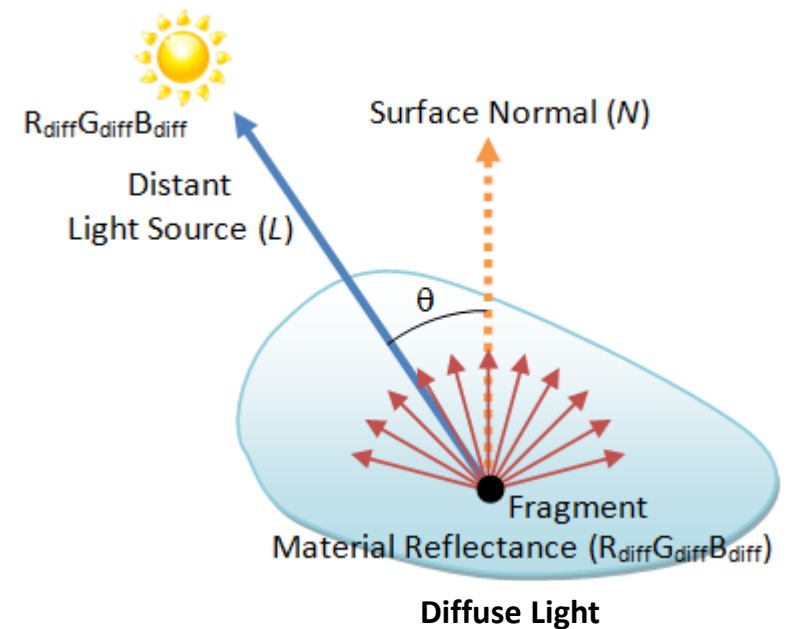


Diffuse Light

- The resultant color can be computed as follows:
 - The strength of the incident light is $\max(L \cdot N, 0)$.
 - We use the $\max()$ function to discard the negative numbers, i.e., when the angle is more than 90 degree.
 - Suppose the light source has color s_{diff} , and the fragment has diffusion reflectance of m_{diff} , the resultant color c is:

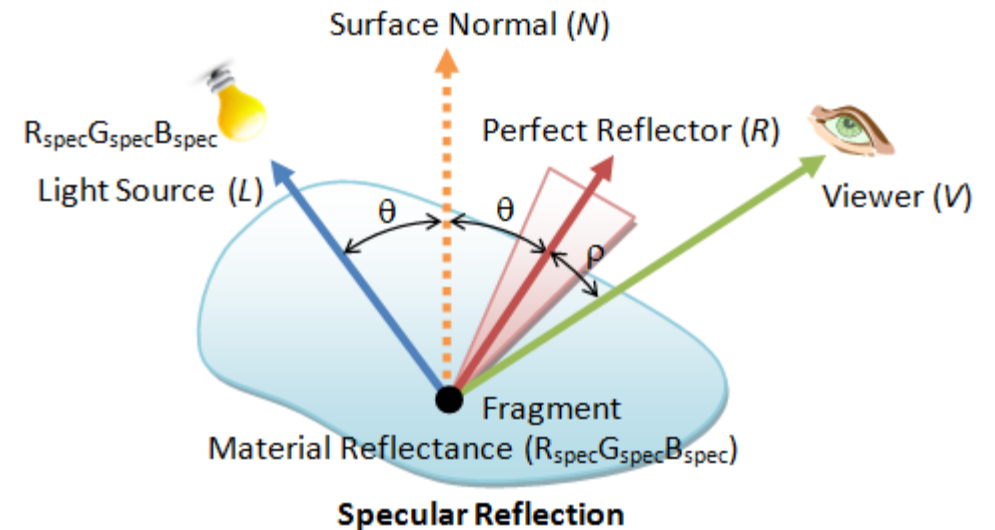
$$c_{\text{diff}} = \max(L \cdot N, 0) s_{\text{diff}} m_{\text{diff}}$$

- The RGB components of the color are computed independently.



Specular Light

- The reflected light is concentrated along the direction of perfect reflector R . What a viewer sees depends on the angle (cosine) between V and R .



- The resultant color c_{spec} due to specular reflection is given by:

$$c_{spec} = \max(R \cdot V, 0)^{sh} s_{spec} m_{spec}$$

- The power sh is known **as the shininess factor**. As sh increases, the light cone becomes narrower (because $R \cdot V \leq 1$), so the highlighted spot becomes smaller.

Ambient, Emissive, and the Resultant Light

- In **ambient lighting**, a constant amount of light is assumed to be applied to every point of the scene

$$C_{\text{amb}} = S_{\text{amb}} m_{\text{amb}}$$

- Some surfaces may emit light. The resultant color is

$$C_{\text{em}} = m_{\text{em}}$$

- The final resultant color is therefore:

$$C_{\text{final}} = C_{\text{diff}} + C_{\text{spec}} + C_{\text{amb}} + C_{\text{ems}}$$

- This is the sum of all four components.

In-Class Demo

The following demo lets you experiment with the material properties that we have discussed so far.

[Material Properties Demo](#)



In-Class Demo

The following demo helps you understand the difference between smooth and flat shading.

[Smooth or Faceted](#)



References

- Interactive Computer Graphics, a Top-Down Approach with WebGL, 7th Edition – Edward Angel and Dave Shreiner, Pearson.
- David J. Eck, Introduction to Computer Graphics, Version 1.3, August 2021. Notes and demos from this source are licensed under a [Creative Commons Attribution-Noncommercial-ShareAlike 4.0 License](#).



Thank you.
DOUGLASCOLLEGE

DOUGLASCOLLEGE

CMPT 2361

Computer Graphics

Shahriar Khosravi, Ph.D.

Lecture 6

Plan for this Lecture

- Light and Material in OpenGL 1.1
- Global Lighting Properties
- Defining Normal Vectors with OpenGL
- Image Textures
- Filtering and MipMaps
- Texture Target and Texture Parameters
- Texture Transformation
- Loading a Texture from Memory
- Texture Objects

Light and Material in OpenGL 1.1

OpenGL Light and Material

- In OpenGL 1.1, the use of light and material must be enabled by calling `glEnable(GL_LIGHTING)`.
- When lighting is disabled, the color of a vertex is simply the current color as set by `glColor*()`.
- When lighting is enabled, the color of a vertex is computed using a mathematical formula that takes into account the lighting of the scene and the material properties that have been assigned to the vertex.
- It is common for lighting to be turned on for rendering some parts of a scene, but turned off for other parts. We will say that some objects are "lit" while others aren't.
- We cannot call `glEnable()` or `glDisable()` between calls to `glBegin()` and `glEnd()`, so it is not possible for part of a primitive to be lit while another part of the same primitive is unlit
- Lighting can be enabled and disabled by calling `glEnable()` and `glDisable()` with parameter `GL_LIGHTING`.

OpenGL Light and Material

- To light a scene, in addition to enabling `GL_LIGHTING`, we must configure at least one source of light.
- For very basic lighting, it is often sufficient to call `glEnable(GL_LIGHT0);`
- This command turns on a directional light that shines from the direction of the viewer into the scene.
- Since it shines from the direction of the viewer, it will illuminate everything that the user can see.
- The light is white, with no specular component (i.e., we will see the diffuse color of objects, without any specular highlights).
- We will discuss later on how to change the characteristics of this light source and how to configure additional sources.
- Before we do that, we must first discuss materials and normal vectors.

Working with Material

- Material properties are vertex attributes (just like color).
- The OpenGL state includes a current value for each of the material properties.
- When a vertex is generated by a call to one of the `glVertex*()` functions, a copy of each of the current material properties is stored, along with the vertex coordinates.
- When a primitive that contains the vertex is rendered, the material properties that are associated with the vertex are used, along with information about lighting, to compute a color for the vertex.
- The front face and back face of a polygon can have different materials, so two sets of material property values must be stored for each vertex.
- The back material isn't usually used unless we turn on **two-sided lighting**.

Working with Material

- For setting the ambient, diffuse, specular, and emission material colors, we must call:

```
void glMaterialfv(int side, int property, float* valueArray);
```

Can be GL_FRONT_AND_BACK, GL_FRONT, or GL_BACK.

which material property is being set.
Can be GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, or GL_AMBIENT_AND_DIFFUSE.

an array containing four **float** numbers, giving the RGBA color components as values in the range from 0.0 to 1.0. An alpha component is required, but it is used only in the case of diffuse color.

- The shininess material property is a single number, and there is a different function for setting its value:

```
void glMaterialf(int side, int property, float value);
```

Can be GL_FRONT_AND_BACK, GL_FRONT, or GL_BACK.

Must be GL_SHININESS.

a **float** in the range 0.0 to 128.0.

Working with Material

- The third parameter in `glMaterialfv()` is a pointer-to-float, so we can store values for several material properties in a single array:

```
float gold[13] = { 0.24725, 0.1995, 0.0745, 1.0,      /* ambient */
                  0.75164, 0.60648, 0.22648, 1.0,      /* diffuse */
                  0.628281, 0.555802, 0.366065, 1.0,    /* specular */
                  50.0,                                /* shininess */
                  };
```

- This array can be used to set all material properties:

```
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, gold);
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, &gold[4]);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, &gold[8]);
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, gold[12]);
```

- `glMaterialfv()` and `glMaterialf()` can be called at any time, including between calls to `glBegin()` and `glEnd()`, so different vertices can have different materials.

Working with Material

- We can use `glColor*()` to work with material as well as regular color by calling `glEnable( GL_COLOR_MATERIAL );`
- With this state enabled, by default, setting the color will also set the current front and back, ambient and diffuse material properties.
- For example, calling `glColor3f(1,0,0 );` with lighting enabled, is the same as `glMaterialfv( GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, arr );` where `arr` is an array of float values `{1.0, 0.0, 0.0}`.
- We can change the material property that tracks the color using:

`void glColorMaterial(side, property);`

Can be `GL_FRONT_AND_BACK`,
`GL_FRONT`, or `GL_BACK`.

Can be `GL_AMBIENT`, `GL_DIFFUSE`,
`GL_SPECULAR`, `GL_EMISSION`,
or `GL_AMBIENT_AND_DIFFUSE`.

- Neither `glEnable()` nor `glColorMaterial()` can be called between calls to `glBegin()` and `glEnd()`, so all of the vertices of a primitive must use the same setting.

Working with Material

- We can use `glColor*()` to work with material as well as regular color by calling `glEnable( GL_COLOR_MATERIAL );`

When `glDrawArrays()` or `glDrawElements()` is used to draw a primitive, the color values for the vertices of the primitive can be taken from a color array, as specified using `glColorPointer()`. **There are no similar arrays for material properties.** However, if a color array is used while lighting is enabled, and if `GL_COLOR_MATERIAL` is also enabled, then the color array will be used as the source for the values of the material properties that are tracking the color.

- Neither `glEnable()` nor `glColorMaterial()` can be called between calls to `glBegin()` and `glEnd()`, so all of the vertices of a primitive must use the same setting.

Defining Normal Vectors with OpenGL

Defining Normal Vectors

- Like color and material, normal vectors are attributes of vertices.
- The OpenGL state includes a current normal vector, which is set using functions in the family `glNormal*()`.
- When a vertex is specified with `glVertex*()`, a copy of the current normal vector is saved as an attribute of the vertex, and it is used as the normal vector for that vertex when the color of the vertex is computed by the lighting equation.
- The normal vector for a vertex must be specified before `glVertex*()` is called for that vertex.
- Functions in the family `glNormal*()` include `glNormal3f()`, `glNormal3d()`, `glNormal3fv()`, and `glNormal3dv()`.
- A "v" means that the values are in an array, "f" means that the values are floats, and "d" means that the values are doubles.
- All normal vectors have three components.

Defining Normal Vectors

- Some examples:


```
glNormal3f(0, 0, 1); // default values
glNormal3d(0.707, 0.707, 0.0);
float normalArray[3] = { 0.577, 0.577, 0.577 };
glNormal3fv(normalArray);
```
- For example, to draw one side of a cube with **flat shading**, say the "top" side, facing in the direction of the positive y-axis:

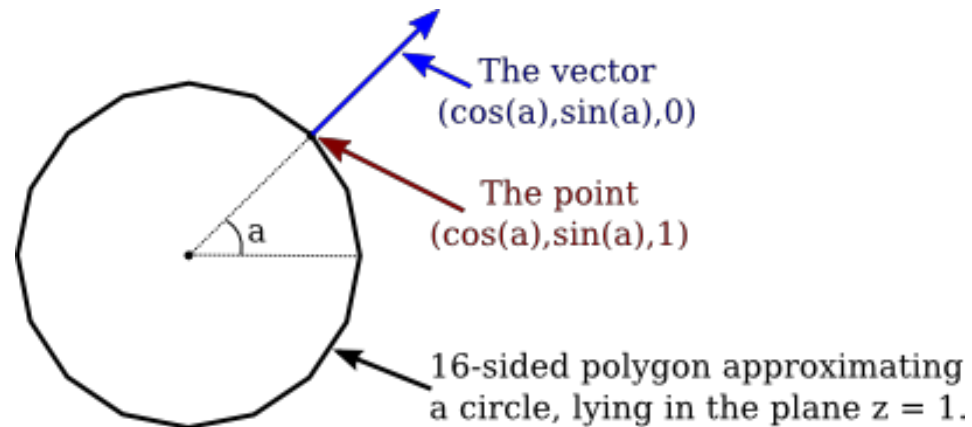
```
glNormal3f(0, 1, 0); // positive y-axis
glBegin(GL_QUADS);
glVertex3f(1, 1, 1);
glVertex3f(1, 1, -1);
glVertex3f(-1, 1, -1);
glVertex3f(-1, 1, 1);
glEnd();
```



- The normal vector should point out of the front face of the polygon, and that the front face is determined by the order in which the vertices are generated.

Defining Normal Vectors

- For **smooth shading**, normal vectors should be chosen perpendicular to the surface, rather than to the polygons that approximate the surface.
- Let's draw the side of a cylinder with radius 1 and height 2, where the center of the cylinder is at $(0,0,0)$ and the axis lies along the z -axis.
- The normal vector points in the same direction as the radius, but its z -coordinate is zero since it points directly out from the side of the cylinder.
- Looking down the z -axis at the top of the cylinder:



Defining Normal Vectors

- To draw the side of the cylinder as a triangle strip, we have to generate pairs of vertices on alternating edges.
- The normal vector is the same for the two vertices in the pair, but it is different for different pairs.

```
glBegin(GL_TRIANGLE_STRIP);  
for (unsigned int i = 0; i <= 16; i++) {  
    double angle = 2 * PI / 16 * i; // i 16-ths of a full circle  
    double x = cos(angle);  
    double y = sin(angle);  
    glNormal3f(x, y, 0); // Normal for both vertices at this angle.  
    glVertex3f(x, y, 1); // Vertex on the top edge.  
    glVertex3f(x, y, -1); // Vertex on the bottom edge.  
}  
glEnd();
```

Defining Normal Vectors – example_03.cpp

- To draw the top and bottom of the cylinder, we need a flat polygon, with the normal vector pointing in the direction (0,0,1) for the top and in the direction (0,0,-1) for the bottom.
- The vertices for the bottom are generated in the opposite order from the vertices for the top, to account for the fact that the top and bottom face in opposite directions.
- Vertices need to be enumerated in counterclockwise order, as seen from the front.

```
glNormal3f(0, 0, 1);
glBegin(GL_TRIANGLE_FAN); // Draw the top, in the plane z = 1.
for (unsigned int i = 0; i < 16; i++) {
    double angle = 2 * PI / 16 * i;
    double x = cos(angle);
    double y = sin(angle);
    glVertex3f(x, y, 1);
}
glEnd();

glNormal3f(0, 0, -1);
glBegin(GL_TRIANGLE_FAN); // Draw the bottom, in the plane z = -1
for (int i = 16; i > 0; i--) {
    double angle = 2 * PI / 16 * i;
    double x = cos(angle);
    double y = sin(angle);
    glVertex3f(x, y, -1);
}
glEnd();
```

Defining Normal Vectors

- The lighting equation assumes that normal vectors are unit normals, that is, that they have length equal to one.
- The default in OpenGL is to use normal vectors as provided, even if they don't have length one, which will give incorrect results.
- We can call `glEnable(GL_NORMALIZE)`; to automatically convert every normal vector into a unit normal that points in the same direction.
- When a geometric transform is applied, normal vectors are transformed along with vertices; this is necessary because a transformation can change the direction in which a surface is facing.
- A scaling transformation can change the length of a normal vector, so even if we provided unit normal vectors, they would not be unit normals after a scaling transformation.
- However, if we have enabled `GL_NORMALIZE`, **the transformed normals will automatically be converted back to unit normals.**



Working with Lights in OpenGL 1.1

Working with Lights

- OpenGL 1.1 supports at least eight light sources, which are identified by the constants `GL_LIGHT0`, `GL_LIGHT1`, ..., `GL_LIGHT7`.
- Each light source can be configured to be either a directional light or a point light, and each light can have its own diffuse, specular, and ambient intensities.
- By default, all of the light sources are disabled.
- To enable a light, call `glEnable(light)`, where `light` is one of the constants `GL_LIGHT0`, `GL_LIGHT1`, etc.
- Simply enabling a light does not give any illumination, except in the case of `GL_LIGHT0`, since all light intensities are zero by default, with the single exception of the diffuse color of light number 0.
- To get any light from the other light sources, we need to change some of their properties.

Working with Lights

- Light properties can be set using:

```
void glLightfv(int light, int property, float* valueArray);
```

Can be one of GL_LIGHT0, GL_LIGHT1, ..., GL_LIGHT7.

Can be GL_DIFFUSE, GL_SPECULAR, GL_AMBIENT, or GL_POSITION.

An array with at least four float numbers for RGBA components between 0 and 1. Alpha component ignored.

- Examples:

```
float blue1[4] = { 0.4, 0.4, 0.6, 1 };  
float blue2[4] = { 0, 0, 0.8, 1 };  
float blue3[4] = { 0, 0, 0.15, 1 };  
glLightfv(GL_LIGHT1, GL_DIFFUSE, blue1);  
glLightfv(GL_LIGHT1, GL_SPECULAR, blue2);  
glLightfv(GL_LIGHT1, GL_AMBIENT, blue3);
```

The GL_POSITION Property

- The `GL_POSITION` property is used both to set whether the light is a point light or a directional light, and to set its position or direction.
- The property value for `GL_POSITION` is an array of four numbers (x, y, z, w) , of which at least one must be non-zero.
- When the fourth value, w , is zero, then the light is directional and the point (x, y, z) specifies the direction of the light (i.e., the light rays shine in the direction of the line from the point (x, y, z) towards the origin).
- In homogeneous coordinates, when w is zero, the source of the light can be considered to be a point at infinity in the direction of (x, y, z) .
- If w is non-zero, then the light is a point light, and it is located at the point $(x/w, y/w, z/w)$.
- Usually, w is 1. The value $(x, y, z, 1)$ gives a point light at (x, y, z) .
- The default position for all lights is $(0, 0, 1, 0)$, representing a directional light shining from the positive direction of the z -axis, towards the negative direction of the z -axis.



The GL_POSITION Property

- The position that is specified for a light is transformed by the modelview transformation that is in effect at the time the position is set using `glLightfv()`.
- Calling `glLightfv()` with the property set to `GL_POSITION` is very much like calling `glVertex*()`. The light position is transformed in the same way that the vertex coordinates would be transformed.
- For example,

```
float position[4] = { 1,2,3,1 };  
glLightfv(GL_LIGHT1, GL_POSITION, position);
```
- puts the light in the same place as:

```
glTranslatef(1, 2, 3);  
float position[4] = { 0,0,0,1 };  
glLightfv(GL_LIGHT1, GL_POSITION, position);
```
- For a directional light, the direction of the light is transformed by the rotational part of the modelview transformation.

The GL_POSITION Property

- There are three basic ways to use light position:
 1. If the position is set before any modelview transformation is applied, then the light is fixed with respect to the viewer. If the viewer moves about in the world, the light moves with the viewer.
 2. If the position is set after the viewing transform has been applied and before any modeling transform is applied, then the position of the light is fixed in world coordinates. It will not move with the viewer, and it will not move with objects in the scene, as though the light is attached to the world.
 3. if the position is set after a modeling transform has been applied, then the light is subject to that modeling transformation. This can be used to make a light that moves around in the scene as the modeling transformation changes. If the light is subject to the same modeling transformation as an object, then the light will move around with that object, as if it is attached to the object.

The GL_POSITION Property – Example





DOUGLAS COLLEGE

Global Lighting Properties

Global Lighting Properties

- The OpenGL lighting system provides three global properties.
 - Global ambient light
 - Two-sided lighting
 - Local viewer's perspective projection option
- Global ambient light is the ambient light that doesn't come from the ambient color property of any light source.
- Global ambient light will be present in the environment even if all of `GL_LIGHT0`, `GL_LIGHT1`, etc. are disabled.
- By default, the global ambient light is black (i.e., its RGB components are all zero). The value can be changed using the function:

```
float ambientLevel[] = { 0.15, 0.15, 0.15, 1 };  
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambientLevel);
```

- The alpha component of the color is usually set to 1, but it is not used for anything.

GL_LIGHT_MODEL_TWO_SIDE

- `GL_LIGHT_MODEL_TWO_SIDE` is used to turn on two-sided lighting.
- When two-sided lighting is off, which is the default, only the front material for a polygon is used; it is used for both the front face and the back face of the polygon.
- Furthermore, the same normal vector is used for both faces. Since those vectors point out of the front face, they don't give the correct result for the back face.
- In effect, the back face looks like it is illuminated by light sources that lie in front of the polygon, but the back face should be illuminated by the lights that lie behind the polygon.
- When two-sided lighting is on, the back material is used on the back face and the direction of the normal vector is reversed when it is used in lighting calculations for the back face.
- We should use two-sided lighting whenever there are back faces that might be visible in the scene.

```
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, 1);
```

Turns it on (1) or off(0).



GL_LIGHT_MODEL_LOCAL_VIEWER

- `GL_LIGHT_MODEL_LOCAL_VIEWER` has to do with the direction from a surface to the viewer in the lighting equation.
- By default, this direction is always taken to point directly out of the screen, which is true for an orthographic projection but is not accurate for a perspective projection.
- If we turn on the local viewer option, the true direction to the viewer is used.
- In practice, the difference is usually not very noticeable.

Image Textures

Image Textures

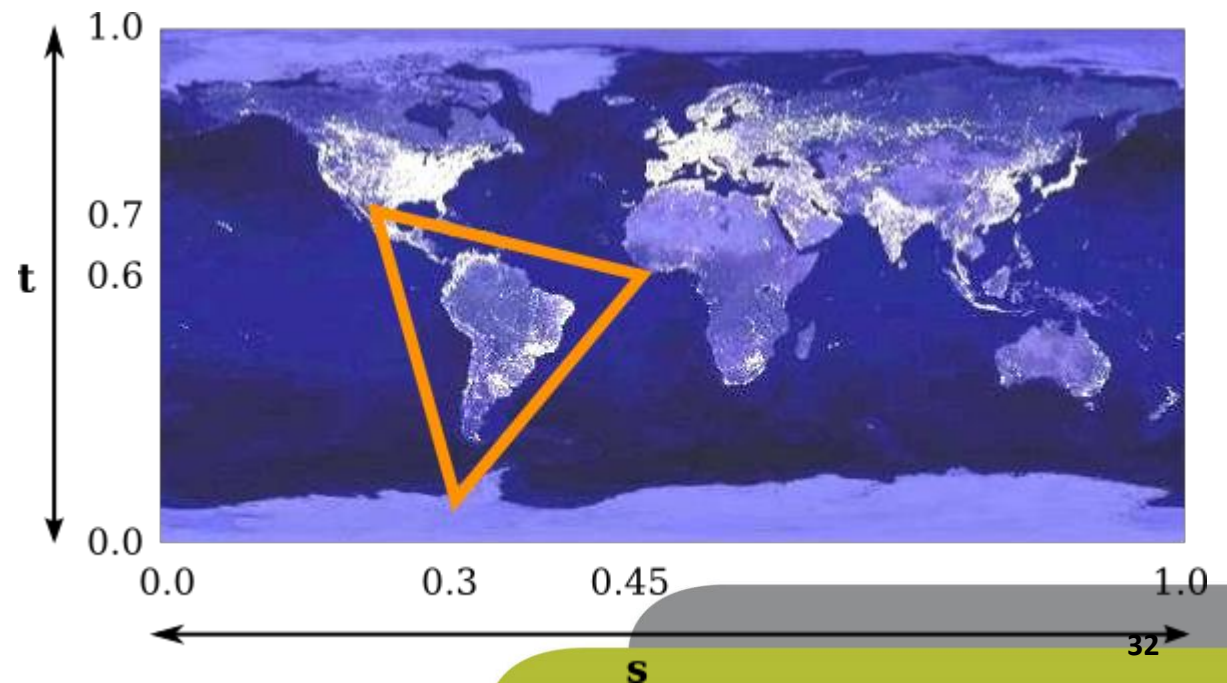
- A texture, is a variation from pixel to pixel within a single primitive. We will consider only one kind of texture: **image textures**.
- An image texture can be applied to a surface to make the color of the surface vary from point to point, something like painting a copy of the image onto the surface.
- An image that is used as a texture should have a width and a height that are powers of two, such as 128, 256, or 512.
- This is a requirement in OpenGL 1.1. The requirement is relaxed in some versions, but it's still a good idea to use power-of-two textures.
- When an image texture is applied to a surface, the default behavior is to multiply the RGBA color components of pixels on the surface by the color components from the image.
- The surface color will be modified by light effects, if lighting is turned on, before it is multiplied by the texture color.
- It is common to use white as the surface color. If a different color is used on the surface, it will add a "tint" to the color from the texture image.

Texture Coordinates

- When a texture is applied to a surface, each point on the surface has to correspond to a point in the texture.
- We need a way to determine how this mapping is computed. For that, the object needs **texture coordinates**.
- OpenGL texture coordinates are specified for each vertex of a primitive. Texture coordinates for points inside the primitive are calculated by interpolating the values from the vertices of the primitive.
- A texture image comes with its own 2D coordinate system. Traditionally, *s* is used for the horizontal coordinate on the image and *t* is used for the vertical coordinate.
- The ***s*** coordinate is a real number that ranges from 0 on the left of the image to 1 on the right, while ***t*** ranges from 0 at the bottom to 1 at the top.
- Values of ***s*** or ***t*** outside of the range 0 to 1 are not inside the image, but such values are still valid as texture coordinates.
- Texture coordinates are not based on pixels. No matter what size the image is, values of ***s*** and ***t*** between 0 and 1 cover the entire image.

Texture Coordinates

- To draw a textured primitive, we need a pair of numbers (s, t) for each vertex. These are the texture coordinates for that vertex and specify which point in the image is mapped to the vertex.
- For example, suppose that we want to apply part of an EarthAtNight image to a triangular primitive.
- Let's say that the area in the image that is to be mapped onto the primitive is the triangle shown here outlined in thick orange:
- The vertices of this area have (s, t) coordinates $(0.3, 0.1)$, $(0.45, 0.6)$, and $(0.25, 0.7)$.
- These coordinates from the image should be used as the texture coordinates for the vertices of the triangular primitive.



Texture Coordinates

- The texture coordinates of a vertex are an attribute of the vertex, just like color, normal vectors, and material properties.
- Texture coordinates are specified by the family of functions `glTexCoord*()`, including the functions `glTexCoord2f(s, t)`, `glTexCoord2d(s, t)`, `glTexCoord2fv(array)`, and `glTexCoord2dv(array)`.
- The OpenGL state includes a current set of texture coordinates, as specified by these functions.
- When you specify a vertex with `glVertex*()`, the current texture coordinates are copied and become an attribute that is associated with the vertex.
- This means that the texture coordinates for a vertex must be specified **before** `glVertex*()` is called.
- Each vertex of a primitive will need a different set of texture coordinates.

Texture Coordinates

- For example, to apply a triangular region in the EarthAtNight image shown previously to the triangle in the xy-plane with vertices at (0,0), (0,1), and (1,0):

```
glNormal3d(0, 0, 1);           // This normal works for all three vertices.
glBegin(GL_TRIANGLES);
glTexCoord2d(0.3, 0.1);        // Texture coords for vertex (0,0)
glVertex2d(0, 0);
glTexCoord2d(0.45, 0.6);       // Texture coords for vertex (0,1)
glVertex2d(0, 1);
glTexCoord2d(0.25, 0.7);       // Texture coords for vertex (1,0)
glVertex2d(1, 0);
glEnd();
```

- What happens if we supply texture coordinates that are not in the range from 0 to 1?
- By default, in OpenGL 1.1, the entire st-plane is filled with copies of the image.
- For example, if the texture coordinates for a square range from 0 to 3 in both directions, instead of 0 to 1, then we will get nine copies of the image on the square (three copies horizontally by three copies vertically).

Texture Coordinates

- To draw a textured primitive using `glDrawArrays()` or `glDrawElements()`, we need to supply the texture coordinates in a vertex array.
- First, enable the use of a texture coordinate array by calling:

```
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
```

- Then, tell OpenGL the location of the data using the function:

```
void glTexCoordPointer(int size, int dataType, int stride, void* array);
```

Will always be 2.

Can be GL_FLOAT,
GL_DOUBLE, or GL_INT.

Usually 0, since there is no
extra data between texture
coordinates in the array.

Pointer to the data, which
must be of the type
indicated by the `dataType`.



DOUGLAS COLLEGE

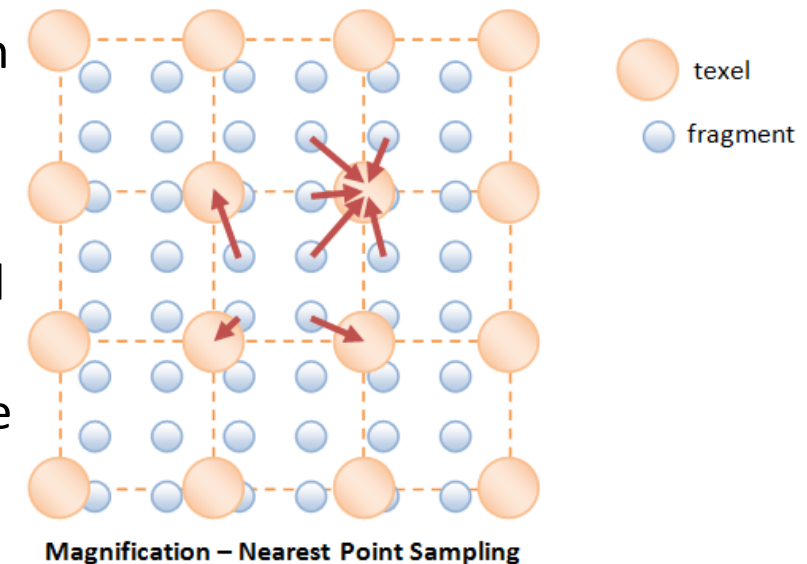
Filtering and MipMaps

Filtering

- When a texture is applied to a surface, the pixels in the texture do not usually match up one-to-one with pixels on the surface.
- Generally, the texture must be stretched or shrunk as it is being mapped onto the surface.
- Sometimes, several pixels in the texture will be mapped to the same pixel on the surface. In this case, the color that is applied to the surface pixel must somehow be computed from the colors of all the texture pixels that map to it.
- This is an example of "**filtering**"; it uses a **minification filter** because the texture is being shrunk.
- When one pixel from the texture covers more than one pixel on the surface, the texture has to be magnified, and we need a **magnification filter**.
- The pixels in a texture are referred to as **texels**, short for "**texture pixel**" or "**texture element**".

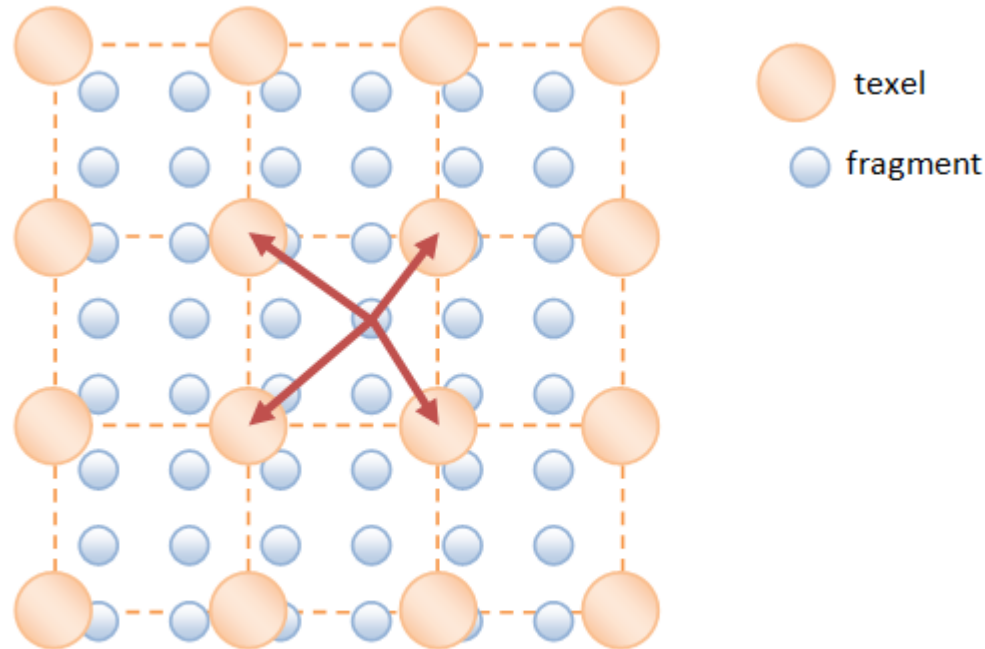
Filtering

- When deciding how to apply a texture to a pixel on a surface, OpenGL must deal with the fact that that the pixel actually contains an infinite number of points, and each point has its own texture coordinates.
- To compute the texture color for the pixel, the easiest approach is to select one point from the pixel, say the point at the center of the pixel.
- OpenGL knows the texture coordinates for that point. Those texture coordinates correspond to one point in the texture, and that point lies in one of the texture's texels.
- The color of that texel could be used as the texture color for the pixel.
- This is called "**nearest texel filtering**." It is very fast, but it does not usually give good results.
- It doesn't take into account the difference in size between the pixels on the surface and the texels in the image.



Filtering

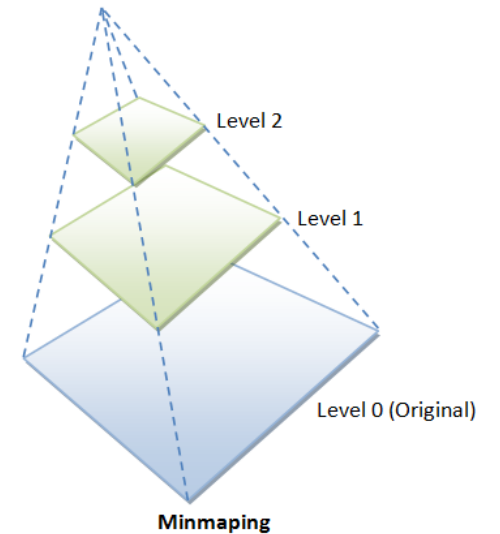
- An improvement on nearest texel filtering is "**linear filtering**," which can take an average of several texel colors to compute the color that will be applied to the surface.
- The problem with linear filtering is that it will be very inefficient when a large texture is applied to a much smaller surface area.
- In this case, many texels map to one pixel, and computing the average of so many texels becomes very inefficient.



Magnification – Bilinear Interpolation

MipMaps

- A **mipmap** for a texture is a scaled-down version of that texture.
- A complete set of mipmaps consists of the full-size texture, a half-size version in which each dimension is divided by two, a quarter-sized version, a one-eighth-sized version, and so on.
- If one dimension shrinks to a single pixel, it is not reduced further, but the other dimension will continue to be cut in half until it too reaches one pixel.
- The final mipmap consists of a single pixel. Here are the first few images in the set of mipmaps for a brick texture:



MipMaps

- Mipmaps are used only for minification filtering. They are essentially a way of pre-computing the bulk of the averaging that is required when shrinking a texture to fit a surface.
- To texture a pixel, OpenGL can first select the mipmap whose texels most closely match the size of the pixel.
- It can then do linear filtering on that mipmap to compute a color, and it will have to average at most a few texels in order to do so.
- In newer versions of OpenGL, we can get OpenGL to generate mipmaps automatically.
- In OpenGL 1.1, if we want to use mipmaps, we must either load each mipmap individually, or we must generate them ourselves.
- The GLU library has a method, `gluBuild2DMipmaps()` that can be used to generate a set of mipmaps for a 2D texture.

Texture Target and Texture Parameters

Texture Parameters

- There are a number of options that apply to textures, to control the details of how textures are applied to surfaces.
- Some of the options can be set using the `glTexParameterf()` function, including two that have to do with filtering.
- OpenGL supports several different filtering techniques for minification and magnification. The filters can be set using `glTexParameteri()`:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, magFilter);
```

The default for the MAG filter is `GL_LINEAR`.

The only options are
`GL_NEAREST` and `GL_LINEAR`.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, minFilter);
```

The default MIN filter is
`GL_NEAREST_MIPMAP_LINEAR`.

Can be `GL_NEAREST`, `GL_LINEAR`,
`GL_NEAREST_MIPMAP_LINEAR`,
`GL_LINEAR_MIPMAP_LINEAR`,
`GL_NEAREST_MIPMAP_NEAREST`, and
`GL_LINEAR_MIPMAP_NEAREST`.

Texture Parameters

- If we are not using mipmaps for a texture, we must still change the minification filter for that texture to `GL_LINEAR`.
- The default MIN filter requires mipmaps, and if mipmaps are not available, then the texture is considered to be improperly formed, and OpenGL ignores it.
- If we don't create mipmaps and if we don't change the minification filter, then our texture will simply be ignored by OpenGL.



Texture Parameters

- There is another pair of texture parameters to control how texture coordinates outside the range 0 to 1 are treated.
- The default behavior is to repeat the texture. The alternative is to "clamp" the texture.
- This means that when texture coordinates outside the range 0 to 1 are specified, those values are forced into that range: Values less than 0 are replaced by 0, and values greater than 1 are replaced by 1.
- Values can be clamped separately in the s and t directions using:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
```

- Passing GL_REPEAT as the last parameter restores the default behavior.
- In this example, left is GL_REPEAT, right is GL_CLAMP in both directions.





DOUGLAS COLLEGE

Texture Transformation

Texture Transformation

- Texture coordinates are no different from vertex coordinates, so they can be transformed in exactly the same way.
- OpenGL maintains a texture transformation as part of its state, along with the modelview and projection transformations.
- The current value of each of the three transformations is stored as a matrix.
- When a texture is applied to an object, the texture coordinates that were specified for its vertices are transformed by the **texture matrix**.
- The transformed texture coordinates are then used to pick out a point in the texture.
- Of course, the default texture transform is the identity transform, which doesn't change the coordinates.

Texture Transformation

- The texture matrix can represent scaling, rotation, translation and combinations of these basic transforms.
- To specify a texture transform, we have to use `glMatrixMode()` to set the matrix mode to `GL_TEXTURE`.
- With this mode in effect, calls to methods such as `glRotate*()`, `glScale*()`, and `glLoadIdentity()` are applied to the texture matrix.
- For example to install a texture transform that scales texture coordinates by a factor of two in each direction, then rotate by 45 degrees around the Z-axis, we could write:

```
glMatrixMode(GL_TEXTURE);  
glLoadIdentity();  
glScalef(2, 2, 1);  
glRotatef(45, 0, 0, 1);  
glMatrixMode(GL_MODELVIEW);
```

Loading a Texture from Memory in C

Loading a Texture

- The OpenGL function for loading image data from the computer's memory into a 2D texture is `glTexImage2D()`:

`glTexImage2D(target, mipmapLevel, internalFormat, width, height, border, format, dataType, pixels);`

Should be
`GL_TEXTURE_2D`.

The value 0 is for loading the main texture; a larger value is used to load an individual mipmap.

`GL_RGB`
or `GL_RGBA`.

The size of the image;
the values should be powers of two.

Should be 0.

`GL_RGB`
or `GL_RGBA`.

Pointer to the start of the color data.

Usually
`GL_UNSIGNED_BYTE`.

- A call to `glTexImage2D()` generally takes the following form:

`glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, imgWidth, imgHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, bitmap);`

- To use the texture, we also have to call:

`glEnable(GL_TEXTURE_2D);`



DOUGLAS COLLEGE

Texture Objects

Texture Objects

- OpenGL 1.1 introduced a new feature called **texture objects** to make texture handling more efficient.
- Texture objects are used when we need to work with several texture images in the same program.
- The usual method for loading texture images, `glTexImage2D()`, transfers data from your program into the graphics card.
- This is an expensive operation, and switching among multiple textures by using this method can seriously degrade a program's performance.
- Texture objects offer the possibility of storing texture data for multiple textures on the graphics card.
- With texture objects, we can switch from one texture object to another with a single, fast OpenGL command: we just have to tell OpenGL which texture object we want to use.
- The graphics card has only a limited amount of memory for storing textures, and we aren't guaranteed that all of our texture objects will actually be stored on the graphics card.

Texture Objects

- Texture objects are managed by OpenGL and the graphics hardware.
- A texture object is identified by an integer ID number.
- To use a texture object, we need to obtain an ID number from OpenGL. This is done with the `glGenTextures()` function:

```
void glGenTextures(int textureCount, int* array);
```

- The first parameter specifies how many IDs we want. The second parameter says where the generated IDs will be stored. It should be an array whose length is at least `textureCount`.

```
int texID[3];  
glGenTextures(3, texID);
```

- We can then use `texID[0]`, `texID[1]`, and `texID[2]` to refer to the textures.
- To work with a specific texture object, we must first call

```
glBindTexture(GL_TEXTURE_2D, texID[i]);
```

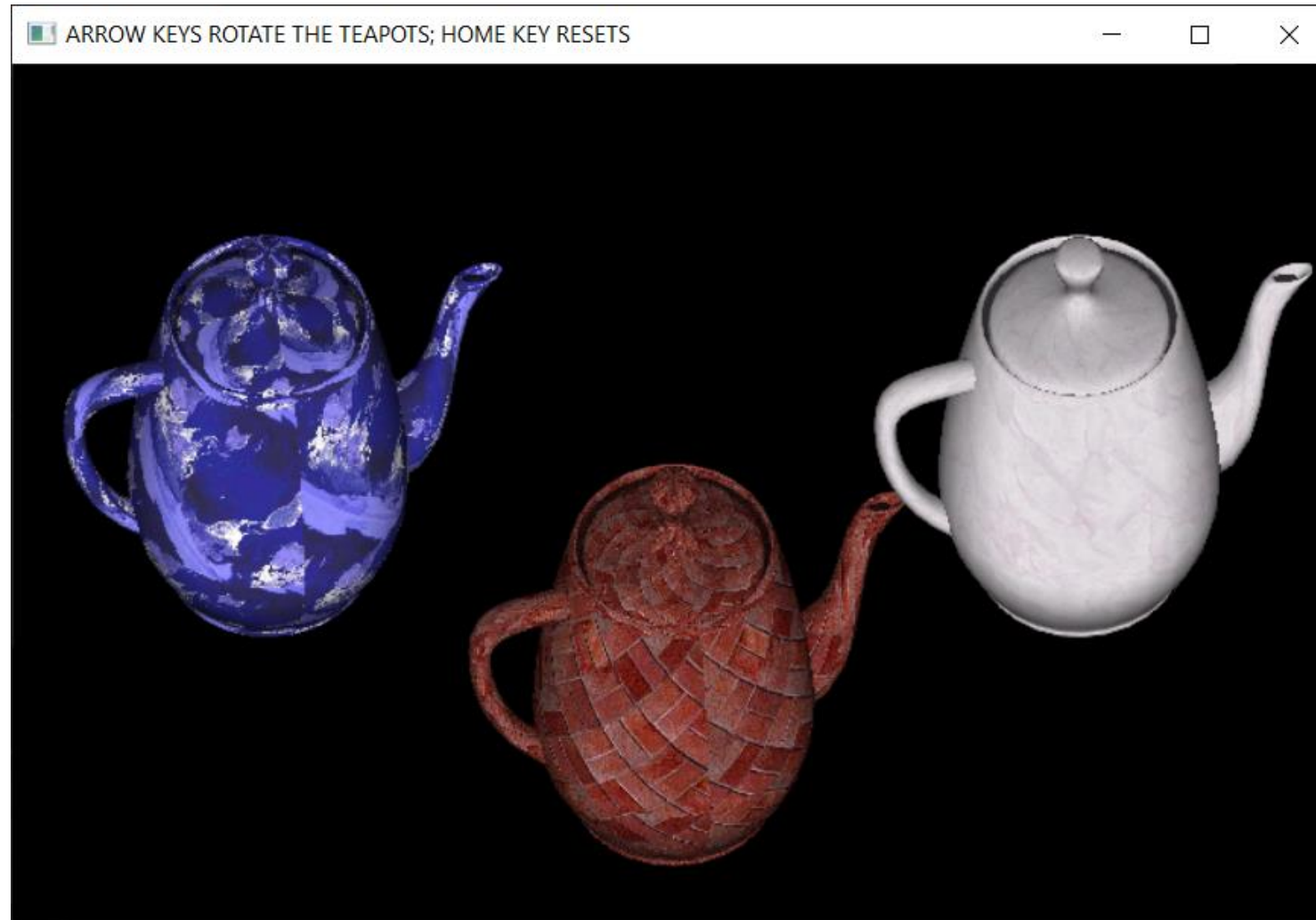
Texture Objects

- After the call to `glBindTexture(GL_TEXTURE_2D, texID[i]);` any use of `glTexParameterf()` and `glTexImage2D()` will be applied to the texture object with ID `texID[i]`.
- When a textured primitive is rendered, the texture that is used is the one that was most recently bound using `glBindTexture()`.

```
// Bind texture #0 for use on a teapot.  
glBindTexture(GL_TEXTURE_2D, texID[0]);  
glPushMatrix();  
glTranslatef(-2.3, 0.3, 0);  
glutSolidTeapot(0.8);  
glPopMatrix();
```

- This would be much more efficient than calling `glTexImage2D()` every time we want to switch textures.

A Texture Example



References

- Interactive Computer Graphics, a Top-Down Approach with WebGL, 7th Edition – Edward Angel and Dave Shreiner, Pearson.
- David J. Eck, Introduction to Computer Graphics, Version 1.3, August 2021. Notes and demos from this source are licensed under a [Creative Commons Attribution-Noncommercial-ShareAlike 4.0 License](#).



Thank you.
DOUGLASCOLLEGE

DOUGLASCOLLEGE

CMPT 2361

Computer Graphics

Shahriar Khosravi, Ph.D.

Lecture 7

Plan for this Lecture

- THREE.js: A 3D Scene Graph API
 - Scene, Renderer, and Camera
 - THREE.Object3D
 - Geometry and Material
 - Color
 - Lines, Line Segments, and Line Loops
 - Meshes
 - Lights
 - Polygonal Meshes and IFS
 - Curves and Surfaces
 - Textures

THREE.js: A 3D Scene Graph API

Three.js: A 3D Scene Graph API

- Three.js is an object-oriented JavaScript library for 3D graphics.
- It is an open-source project created by Ricardo Cabello (who goes by the handle "mr.doob", <https://mrdoob.com/>), with contributions from other programmers.
- It is one of the most popular open-source JavaScript library for 3D web applications.
- Three.js uses concepts that we are already familiar with, such as geometric objects, transformations, lights, materials, textures, and cameras. But it also has additional features that build on the power and flexibility of WebGL.
- The core features of three.js are defined in a single large JavaScript file named "three.js". There is also a smaller "minified" version, three.min.js, that contains the same definitions in a format that is not meant to be human-readable.
- To use three.js on a web page, we can load either script with a `<script>` element on the page. For example, assuming that three.min.js is in the same folder as the web page, then the script element would be:

```
<script src="three.min.js"></script>
```



DOUGLAS COLLEGE

Scene, Renderer, and Camera

Scene, Renderer, and Camera

- Three.js works with the HTML `<canvas>` element.
- In almost all web browsers, in addition to its 2D Graphics API, a canvas also supports drawing in 3D using WebGL, which is used by three.js.
- Three.js is an object-oriented scene graph API. The basic procedure is to build a scene graph out of three.js objects, and then to render an image of the scene it represents.
- Animation can be implemented by modifying properties of the scene graph between frames.
- The three.js library is made up of a large number of classes. Three of the most basic are `THREE.Scene`, `THREE.Camera`, and `THREE.WebGLRenderer`.
- A three.js program will need at least one object of each type.
- Those objects are often stored in global variables:

```
let scene, renderer, camera;
```

Scene, Renderer, and Camera

- Almost all of the three.js classes and constants that we will use are properties of an object named `THREE`, and their names begin with "`THREE.`".
- A **Scene** object is a holder for all the objects that make up a 3D world, including lights, graphical objects, and possibly cameras. It acts as a root node for the scene graph.
- A **Camera** is a special kind of object that represents a viewpoint from which an image of a 3D world can be made. It represents a combination of a viewing transformation and a projection.
- A **WebGLRenderer** is an object that can create an image from a scene graph.
- The scene is the simplest of the three objects. A scene can be created as an object of type `THREE.Scene` using a constructor with no parameters:

```
scene = new THREE.Scene();
```

- The function `scene.add(item)` can be used to add cameras, lights, and graphical objects to the scene. The function `scene.remove(item)`, which removes an item from the scene, is also useful.



DOUGLAS COLLEGE

Camera

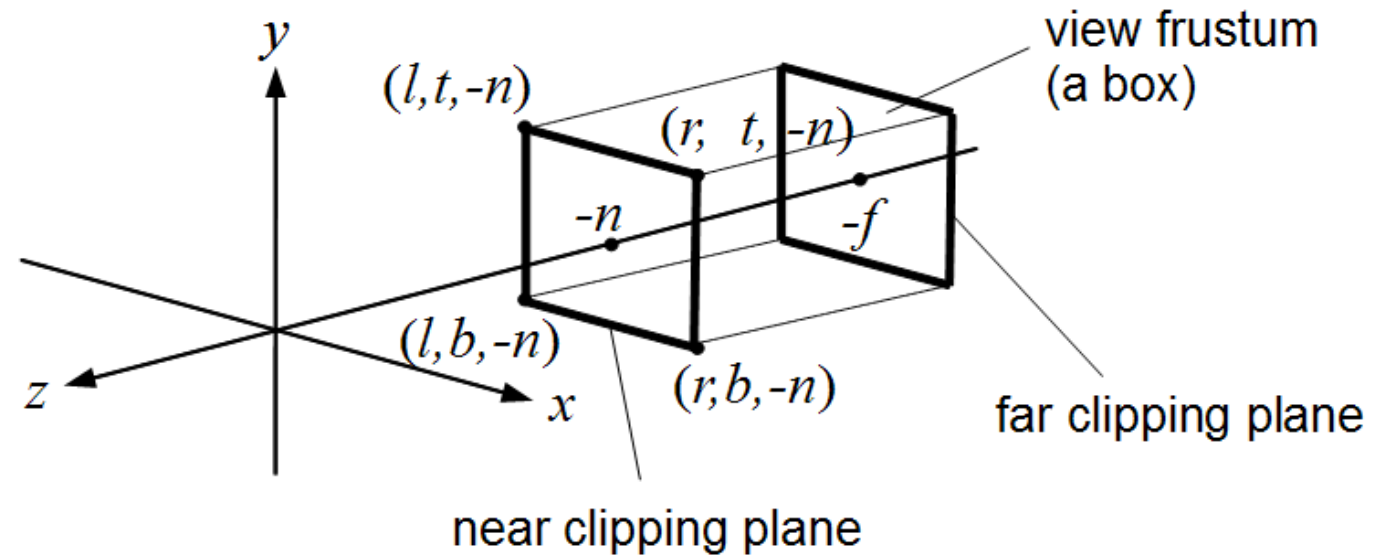
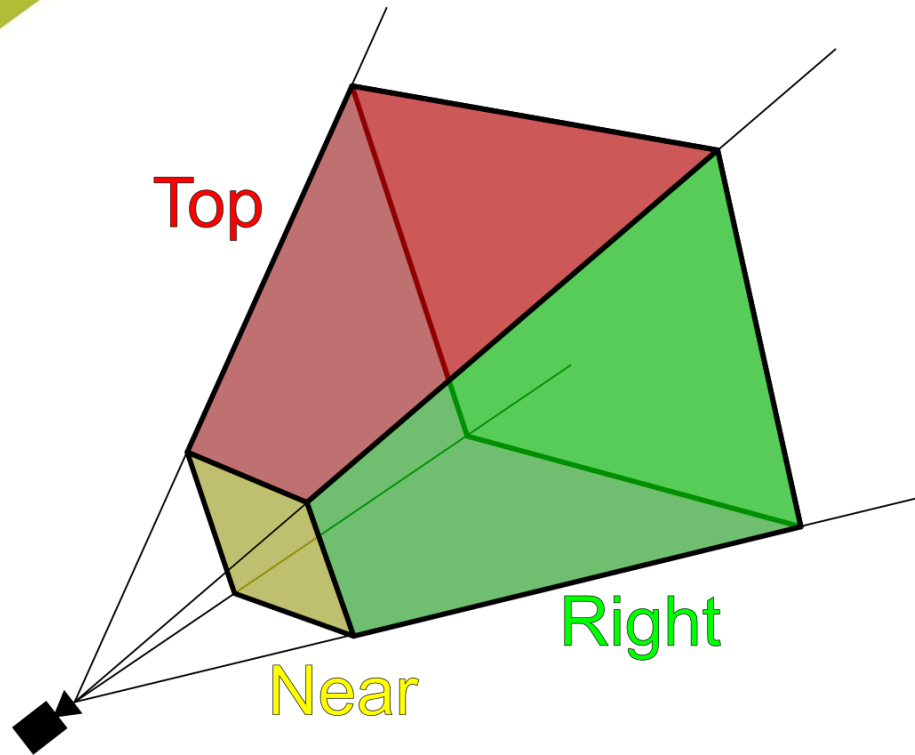
Camera

- There are two kinds of camera, one using **orthographic projection** and one using **perspective projection**.
- They are represented by classes `THREE.OrthographicCamera` and `THREE.PerspectiveCamera`, which are subclasses of `THREE.Camera`.
- The constructors specify the projection, using parameters that are familiar from OpenGL:

```
camera = new THREE.OrthographicCamera( left, right, top, bottom, near, far );
```

- The parameters for the orthographic camera specify the x, y, and z limits of the view volume, in eye coordinates (i.e., in a coordinate system in which the camera is at (0,0,0) looking in the direction of the negative z-axis, with the y-axis pointing up in the view.)
- The **near** and **far** parameters give the z-limits in terms of distance from the camera. For an orthographic projection, **near** can be negative, putting the "near" clipping plane in back of the camera.
- The parameters are the same as for the OpenGL function `glOrtho()`, except for reversing the order of the two parameters that specify the top and bottom clipping planes.

Camera



Camera

- The parameters for the perspective camera come from the function `gluPerspective()` in OpenGL's GLU library.

```
camera = new THREE.PerspectiveCamera( fieldOfViewAngle, aspect, near, far );
```

- `fieldOfViewAngle` determines the vertical extent of the view volume, given as an angle measured in degrees.
- Parameter `aspect` is the ratio between the horizontal and vertical extents; it should usually be set to the width of the canvas divided by its height.
- The `near` and `far` parameters give the z-limits on the view volume as distances from the camera. For a perspective projection, both must be positive, with `near` less than `far`.

Typical code for creating a perspective camera would be:

```
camera = new THREE.PerspectiveCamera( 45, canvas.width/canvas.height, 1, 100 );
```

- where `canvas` holds a reference to the `<canvas>` element where the image will be rendered.

Renderer

Renderer

- A **renderer** is an instance of the class `THREE.WebGLRenderer`.
- Its constructor has one parameter, which is a JavaScript object containing settings that affect the renderer.
- The settings to specify are **canvas**, which tells the renderer where to draw, and **antialias**, which asks the renderer to use antialiasing:

```
renderer = new THREE.WebGLRenderer( {  
    canvas: theCanvas,  
    antialias: true  
} );
```

- Here, **theCanvas** would be a reference to the `<canvas>` element where the renderer will display the images that it produces.
- The main thing to do with a renderer is render an image. For that, we also need a scene and a camera. To render an image of a given scene from the point of view of a given camera:

```
renderer.render( scene, camera );
```




DOUGLAS COLLEGE

THREE.Object3D

THREE.Object3D

- `THREE.Object3D` is the base class for most objects in three.js and provides a set of properties and methods for manipulating objects in 3D space.
- A THREE.js scene graph is made up of objects of type `THREE.Object3D` including objects that belong to subclasses of that class.
- Cameras, lights, and visible objects are all represented by subclasses of `Object3D`. In fact, `THREE.Scene` itself is also a subclass of `Object3D`.
- An `Object3D`, `obj`, has an associated transformation, which is given by properties `obj.scale`, `obj.rotation`, and `obj.position`.
- These properties represent a modeling transformation to be applied to the object and its children when the object is rendered.
- The object is first scaled, then rotated, then translated according to the values of these properties.

THREE.Object3D

- The values of `obj.scale` and `obj.position` are objects of type `THREE.Vector3`.
- A `Vector3` represents a vector or point in three dimensions. There are also similar classes `THREE.Vector2` and `THREE.Vector4` for vectors in 2 and 4 dimensions.
- A `Vector3` object can be constructed from three numbers that give the coordinates of the vector:

```
let v = new THREE.Vector3( 17, -3.14159, 42 );
```

- This object has properties `v.x`, `v.y`, and `v.z` representing the coordinates.
- The properties can be set individually; for example: `v.x = 10`.
- They can also be set all at once, using the method `v.set(x,y,z)`.
- The `Vector3` class also has many methods implementing vector operations such as addition, dot product, and cross product.
- For an `Object3D`, the properties `obj.scale.x`, `obj.scale.y`, and `obj.scale.z` give the amount of scaling of the object in the x, y, and z directions. The default values are 1.

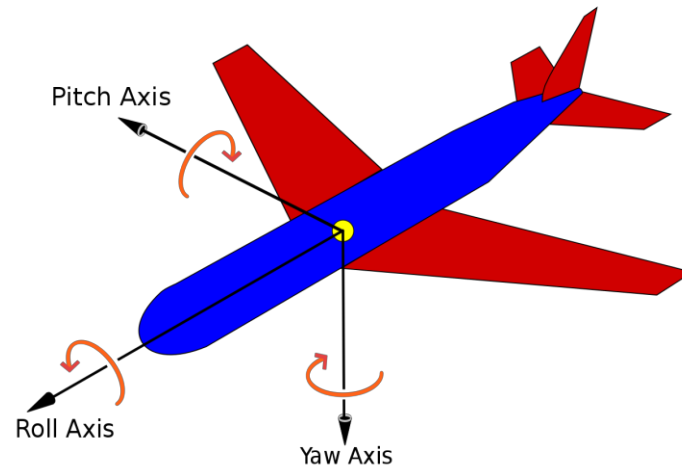
THREE.Object3D

- Calling `obj.scale.set(2,2,2);` means that the object will be subjected to a uniform scaling factor of 2 when it is rendered.
- Setting `obj.scale.y = 0.5;` will shrink it to half-size in the y-direction only.
- Similarly, the properties `obj.position.x`, `obj.position.y`, and `obj.position.z` give the translation amounts that will be applied to the object in the x, y, and z directions when it is rendered.
- For example, since a camera is an Object3D, setting `camera.position.z = 20;` means that the camera will be moved from its default position at the origin to the point (0,0,20) on the positive z-axis.
- This **modeling transformation** on the camera becomes a **viewing transformation** when the camera is used to render a scene.



THREE.Object3D

- The object `obj.rotation` has properties `obj.rotation.x`, `obj.rotation.y`, and `obj.rotation.z` that represent rotations about the x-, y-, and z-axes.
- The angles are measured in **radians**.
- The object is rotated first about the x-axis, then about the y-axis, then about the z-axis.
- The value of `obj.rotation` is **not** a vector. Instead, it belongs to a similar type, `THREE.Euler`, and the angles of rotation are called **Euler angles**.
- The Euler angles are three angles introduced by Leonhard Euler to describe the orientation of a rigid body with respect to a fixed coordinate system.





DOUGLAS COLLEGE

Geometry and Material

Geometry and Material

- A visible object in three.js is made up of either points, lines, or triangles.
- An individual object corresponds to an OpenGL primitive such as `GL_POINTS`, `GL_LINES`, or `GL_TRIANGLES`.
- There are five classes to represent these possibilities:
 - `THREE.Points` for points;
 - `THREE.Mesh` for triangles;
 - `THREE.Line`, which uses the `GL_LINE_STRIP` primitive;
 - `THREE.LineSegments`, which uses the `GL_LINES` primitive; and
 - `THREE.LineLoop`, which uses the `GL_LINE_LOOP` primitive.
- A visible object is made up of some geometry plus a material that determines the appearance of that geometry.
- In three.js, the geometry and material of a visible object are themselves represented by JavaScript classes `THREE.BufferGeometry` and `THREE.Material`.

Geometry and Material

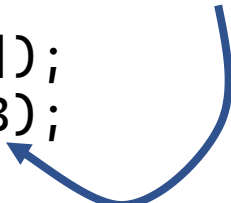
- An object of type `THREE.BufferGeometry` can store vertex coordinates and their attributes. The vertex coordinates are also considered to be an "attribute" of the geometry.
- These values must be stored in a form suitable for use with the OpenGL functions `glDrawArrays()` and `glDrawElements()`.
- For JavaScript, this means that they must be stored in **typed arrays**. A typed array is similar to a normal JavaScript array, except that its length is fixed and it can only hold numerical values of a certain type.
 - For example, a `Float32Array` holds 32-bit floating point numbers, and a `Uint16Array` holds unsigned 16-bit integers.
- A typed array can be created with a constructor that specifies the length of the array:
`vertexCoords = new Float32Array(300); // Space for 300 numbers.`
- Alternatively, the constructor can take an ordinary JavaScript array of numbers as its parameter. Here, the length of `data` is 6, and it contains a copy of the input elements:
`data = new Float32Array( [ 1.3, 7, -2.89, 0, 3, 5.5 ] );`

Geometry and Material

- Specifying the vertices for a **BufferGeometry** is a multistep process.
 - We need to create a typed array containing the coordinates of the vertices.
 - Then we need to wrap that array inside an object of type **THREE.BufferAttribute**.
 - Finally, we can add the attribute to the geometry. Here is an example:

```
let vertexCoords = new Float32Array([ 0,0,0, 1,0,0, 0,1,0 ]);  
let vetexAttrib = new THREE.BufferAttribute(vertexCoords, 3);  
let geometry = new THREE.BufferGeometry();  
geometry.setAttribute( "position", vertexAttrib );
```

An integer that tells three.js the number of coordinates of each vertex.



- As for the **setAttribute()** function, a **BufferGeometry** can have attributes specifying color, normal vectors, and texture coordinates, as well as other custom attributes.
- The first parameter to **setAttribute()** is the name of the attribute. Here, "position" is the name of the attribute that specifies the coordinates, or position, of the vertices.

Geometry and Material

- To specify a color for each vertex, we can put the RGB components of the colors into a `Float32Array`, and use it to specify a value for the `BufferGeometry` attribute named "color".
- Suppose that we want to represent a primitive of type `GL_POINTS`, using a three.js object of type `THREE.Points`.
- Let's say we want 10000 points placed at random inside the unit sphere, where each point can have its own random color.
- How do we achieve that?

Geometry and Material

- To sp
Floa
"col
- Supp
of ty
- Let's
can h
- How

```

let pointsBuffer = new Float32Array( 30000 ); // 3 numbers per vertex.
let colorBuffer = new Float32Array( 30000 );
let i = 0;
while ( i < 10000 ) {
    let x = 2*Math.random() - 1;
    let y = 2*Math.random() - 1;
    let z = 2*Math.random() - 1;
    if ( x*x + y*y + z*z < 1 ) {
        // only use points inside the unit sphere
        pointsBuffer[3*i] = x;
        pointsBuffer[3*i+1] = y;
        pointsBuffer[3*i+2] = z;
        colorBuffer[3*i] = 0.25 + 0.75*Math.random();
        colorBuffer[3*i+1] = 0.25 + 0.75*Math.random();
        colorBuffer[3*i+2] = 0.25 + 0.75*Math.random();
        i++;
    }
}
let pointsGeom = new THREE.BufferGeometry();
pointsGeom.setAttribute("position",
    new THREE.BufferAttribute(pointsBuffer,3));
pointsGeom.setAttribute("color",
    new THREE.BufferAttribute(colorBuffer,3));
  
```

nto a
ite named

e.js object

ach point

Geometry and Material

- In three.js, to make some geometry into a visible object, we also need an appropriate material.
- For example, for an object of type `THREE.Points`, we can use a material of type `THREE.PointsMaterial`, which is a subclass of `Material`.
- The material can specify the color and the size of the points, among other properties.

```
let pointsMat = new THREE.PointsMaterial( {  
    color: "yellow",  
    size: 2,  
    sizeAttenuation: false  
} );
```

- When `sizeAttenuation` is `false`, the size is given in pixels; if `true`, then size represents the size in world coordinates and the point is scaled to reflect distance from the viewer.
- If the `color` is omitted, a default value of white is used. The default for `size` is 1 and for `sizeAttenuation` is `true`.
- The parameter to the constructor can be omitted entirely, to use all the defaults. A `PointsMaterial` is not affected by lighting; it simply shows the color specified by its color property.

Geometry and Material

- It is also possible to assign values to properties of the material after the object has been created:

```
let pointsMat = new THREE.PointsMaterial();
pointsMat.color = new THREE.Color("yellow");
pointsMat.size = 2;
pointsMat.sizeAttenuation = false;
```

- Note that the color is set as a value of type `THREE.Color`, which is constructed from a string, **"yellow"**.
- When the color property is set in the material constructor, the same conversion of string to color is done automatically.
- Once we have the geometry and the material, we can use them to create the visible object, of type `THREE.Points`, and add it to a scene:

```
let sphereOfPoints = new THREE.Points( pointsGeom, pointsMat );
scene.add( sphereOfPoints );
```

Geometry and Material

- What if we wanted each point to have its own color?
- Recall that the colors for the points are stored in the geometry, not in the material.
- We have to tell the material to use the colors from the geometry, not the material's own color property. This is done by setting the value of the material property `vertexColors` to `true`:

```
let pointsMat = new THREE.PointsMaterial( {  
    color: "white",  
    size: 2,  
    sizeAttenuation: false,  
    vertexColors: true  
} );
```

- White is used here as the material color because the vertex colors are **multiplied by the material color**, not simply substituted for it.

In-Class Demo

The following demo shows a point cloud. You can control whether the points are all yellow or are randomly colored. You can animate the points, and you can control the size and number of points. Note that points are rendered as squares.

[Three.js Points Demo](#)





DOUGLAS COLLEGE

Color

Color

- Colors in three.js can be represented by values of type `THREE.Color`.
- The class `THREE.Color` represents an RGB color. A Color object `c` has properties `c.r`, `c.g`, and `c.b` for the red, blue, and green color components as floating point numbers in the range from 0.0 to 1.0.
- There is no alpha component; three.js handles transparency separately from color.
- The `THREE.Color` constructor can take three parameters giving the RGB components as real numbers in the range 0.0 to 1.0.
- It can also take a single string parameter giving the color as a CSS color string, like those used in the 2D canvas graphics API; examples include "white", "red", "rgb(255, 0, 0)", and "#FF0000".
- It can also take a single integer parameter in which each color component is given as an eight-bit field in the integer.
 - Usually, an integer that is used to represent a color in this way is written as a hexadecimal literal, beginning with "0x". Examples include `0xff0000` for red, `0x00ff00` for green, `0x0000ff` for blue, and `0x007050` for a dark blue-green.

Color

- Here are some examples of using color constructors:

```
let c1 = new THREE.Color("skyblue");  
let c2 = new THREE.Color(1,1,0); // yellow  
let c3 = new THREE.Color(0x98fb98); // pale green
```

- In many contexts, such as the **THREE.Points** constructor, three.js will accept a string or integer where a color is required; the string or integer will be fed through the Color constructor.
- As another example, a **WebGLRenderer** object has a “ClearColor” property that is used as the background color when the renderer renders a scene. This property could be set using any of the following commands:

```
renderer.setClearColor( new THREE.Color(0.6, 0.4, 0.1) );  
renderer.setClearColor( "darkgray" );  
renderer.setClearColor( 0x99BBEE );
```

Lines, Line Segments, and Line Loops

Lines

- An object of type `THREE.Line` represents a line strip (what would be a primitive of the type called `GL_LINE_STRIP` in OpenGL).
- To get the same strip of connected line segments, plus a line back to the starting vertex, we can use an object of type `THREE.LineLoop`.
- For the outline of a triangle, for example, we could provide a `BufferGeometry` holding coordinates for three points and use a `LineLoop`.
- For lines, the material can be represented by an object of type `THREE.LineBasicMaterial`.
- The "Basic" in `LineBasicMaterial` means that this material uses basic colors that do not require lighting to be visible and are not affected by lighting.
- The parameter for the constructor is a JavaScript object, whose properties can include `color` and `linewidth`:

```
let lineMat = new THREE.LineBasicMaterial( {  
    color: 0xA000A0, // purple; the default is white  
    linewidth: 2      // 2 pixels; the default is 1  
} );
```

Lines

- As with points, it is possible to specify a different color for each purpose by adding a "color" attribute to the geometry and setting the value of the **vertexColors** material property to true.
- Here is a complete example that makes a triangle with vertices colored red, green, and blue:



```
let positionBuffer = new Float32Array([
    -2, -2,    // Coordinates for first vertex.
     2, -2,    // Coordinates for second vertex.
     0,  2,    // Coordinates for third vertex.
]);
let colorBuffer = new Float32Array([
    1, 0, 0,   // Color for first vertex (red).
    0, 1, 0,   // Color for second vertex (green).
    0, 0, 1,   // Color for third vertex (blue).
]);
let lineGeometry = new THREE.BufferGeometry();
lineGeometry.setAttribute(
    "position",
    new THREE.BufferAttribute(positionBuffer, 2)
);
lineGeometry.setAttribute(
    "color",
    new THREE.BufferAttribute(colorBuffer, 3)
);
let lineMaterial = new THREE.LineBasicMaterial( {
    linewidth: 3,
    vertexColors: true
} );
let triangle = new THREE.LineLoop( lineGeometry, lineMaterial );
scene.add(triangle);
```



DOUGLAS COLLEGE

Meshes

Meshes

- A mesh object in three.js corresponds to the OpenGL primitive `GL_TRIANGLES`. The geometry object for a mesh must specify which vertices are part of which triangles.
- THREE.js comes with classes to represent common mesh geometries, such as a sphere, a cylinder, and a torus.
- For these built-in classes, we just need to call a constructor to create the appropriate geometry.
- For example, the class `THREE.CylinderGeometry` represents the geometry for a cylinder, and its constructor takes the form:

```
new THREE.CylinderGeometry(radiusTop, radiusBottom, height,  
                           radiusSegments, heightSegments, openEnded,  thetaStart, thetaLength)
```

All parameters are optional. The default value for the first three parameters is one. The default for `radiusSegments` is 8. Leaving out the last three parameters will give a complete cylinder, closed at both ends.



A Boolean value that indicates whether the top and bottom of the cylinder are to be drawn; use the value `true` to get an open-ended tube.

These allow us to make a partial cylinder. Their values are given as angles, measured in **radians**, about the y-axis.

Meshes

- Other standard mesh geometries are similar. Here are some constructors, listing all parameters (most parameters are optional).

```
new THREE.BoxGeometry(width, height, depth,  
                      widthSegments, heightSegments, depthSegments)
```

```
new THREE.PlaneGeometry(width, height, widthSegments, heightSegments)
```

```
new THREE.RingGeometry(innerRadius, outerRadius, thetaSegments, phiSegments,  
                      thetaStart, thetaLength)
```

```
new THREE.ConeGeometry(radiusBottom, height, radiusSegments,  
                      heightSegments, openEnded, thetaStart, thetaLength)
```

```
new THREE.SphereGeometry(radius, widthSegments, heightSegments,  
                      phiStart, phiLength, thetaStart, thetaLength)
```

```
new THREE.TorusGeometry(radius, tube, radialSegments, tubularSegments, arc)
```


Meshes and Materials

- To create a mesh object, we need a material as well as a geometry.
- There are several kinds of material suitable for mesh objects, including `THREE.MeshBasicMaterial`, `THREE.MeshLambertMaterial`, and `THREE.MeshPhongMaterial`.
- A `MeshBasicMaterial` represents a color that is not affected by lighting; it looks the same whether or not there are lights in the scene, and it is not shaded, giving it a flat rather than 3D appearance.
- The other two classes represent materials that need to be lit to be seen. They implement models of lighting known as **Lambert shading** and **Phong shading**.
- The major difference is that `MeshPhongMaterial` has a specular color but `MeshLambertMaterial` does not. Both can have diffuse and emissive colors.
- A Lambert material lacks specular and shininess, and a basic mesh material has only the color parameter.


```
let mat = new THREE.MeshPhongMaterial( {
    color: 0xbbbb00,    // reflectivity for diffuse and ambient light
    emissive: 0,        // emission color; this is the default (black)
    specular: 0x070707, // reflectivity for specular light
    shininess: 50       // controls size of specular highlights
  } );
```

Meshes and Materials

- There are a few other material properties that you might need to set in the constructor. Except for `flatShading`, these apply to all three kinds of mesh material:
- `vertexColors` – a boolean property that can be set to true to use vertex colors from the geometry. The default is `false`.
- `wireframe` – a boolean value that indicates whether the mesh should be drawn as a wireframe model, showing only the outlines of its faces. The default is `false`. A `true` value works best with `MeshBasicMaterial`.
- `wireframeLinewidth` – the width of the lines used to draw the wireframe, in pixels. The default is 1. (Non-default values might not be respected.)

Meshes and Materials

- **visible** – a boolean value that controls whether the object on which it is used is rendered or not, with a default of **true**.
- **side** – has value **THREE.FrontSide**, **THREE.BackSide**, or **THREE.DoubleSide**, with the default being **THREE.FrontSide**. This determines whether faces of the mesh are drawn or not, depending on which side of the face is visible. With the default value, **THREE.FrontSide**, a face is drawn only if it is being viewed from the front. **THREE.DoubleSide** will draw it whether it is viewed from the front or from the back, and **THREE.BackSide** only if it is viewed from the back. For closed objects, such as a cube or a complete sphere, the default value makes sense, at least as long as the viewer is outside of the object. For a plane, an open tube, or a partial sphere, the value should be set to **THREE.DoubleSide**.
- **flatShading** – a boolean value, with the default being false. This works only for **MeshPhongMaterial**. For an object that is supposed to look "faceted," with flat sides, it is important to set this property to **true**. That would be the case, for example, for a cube or for a cylinder with a small number of sides.

Meshes and Materials

- As an example, let's make a shiny, blue-green, open, five-sided tube with flat sides:

```
let mat = new THREE.MeshPhongMaterial( {  
    color: 0x0088aa,  
    specular: 0x003344,  
    shininess: 100,  
    flatShading: true, // for flat-looking sides  
    side: THREE.DoubleSide // for drawing the inside of the tube  
} );  
let geom = new THREE.CylinderGeometry(3,3,10,5,1,true);  
let obj = new THREE.Mesh(geom,mat);  
scene.add(obj);
```

Meshes, Materials, and the Depth Test

- In three.js, we can apply polygon offset to a material. For example:

```
mat = new THREE.MeshLambertMaterial({  
  polygonOffset: true,  
  polygonOffsetUnits: 1,  
  polygonOffsetFactor: 1,  
  color: "yellow",  
  side: THREE.DoubleSide  
});
```

- The settings shown here for `polygonOffset`, `polygonOffsetUnits`, and `polygonOffsetFactor` will increase the depth of the object that uses this material slightly, so that it doesn't interfere with the wireframe version of the same object.
- We don't always need to make new materials and geometries to make new objects. We can reuse the same materials and geometries in multiple objects.



DOUGLAS COLLEGE

Lights

Lights

- Three.js has several classes to represent lights. Light classes are subclasses of `THREE.Object3D`.
- A light object can be added to a scene and will then illuminate objects in the scene. We will look at **directional lights**, **point lights**, **ambient lights**, and **spotlights**.
- The class `THREE.DirectionalLight` represents light that shines in parallel rays from a given direction, like the light from the sun.
- The **position** property of a directional light gives the direction from which the light shines.
- This is the same **position** property, of type `Vector3`, that all scene graph objects have, but the meaning is different for directional lights.
- Light shines from the given position towards the origin. The default position is the vector (0,1,0), which gives a light shining down the y-axis. The constructor for this class has two parameters:
`new THREE.DirectionalLight( color, intensity )`

A `THREE.Color` object, or as a hexadecimal integer, or as a CSS color string.

A non-negative number that controls the brightness of the light, between 0 and 1.

Lights

- Suppose that we have a camera on the positive z-axis, looking towards the origin, and we would like a light that shines in the same direction that the camera is looking. We can use a directional light whose position is on the positive z-axis:

```
let light = new THREE.DirectionalLight(); // default white light
light.position.set( 0, 0, 1 );
scene.add(light);
```

- The class `THREE.PointLight` represents a light that shines in all directions from a point.
- The location of the point is given by the light's `position` property. The constructor has three optional parameters:

```
new THREE.PointLight( color, intensity, cutoff )
```

- The `cutoff` is a non-negative number. When zero (default), the illumination from the light extends to infinity, and intensity does not decrease with distance. If greater than zero, then the intensity falls from a maximum value at the light's position down to an intensity of zero at a distance of cutoff from the light (**attenuation**).

Lights

- The `THREE.AmbientLight` class exists to add ambient light to a scene. An ambient light has only a color:

```
new THREE.AmbientLight( color )
```

- For example, suppose that we would like a yellowish point light at (10,30,15) whose illumination falls off with distance from that point, out to a distance of 100 units.
- We also want to add a bit of yellow ambient light to the scene:

```
let light = new THREE.PointLight( 0xffffcc, 1, 100 );  
light.position.set( 10, 30, 15 );  
scene.add(light);  
scene.add( new THREE.AmbientLight(0x111100) );
```

Lights

- The fourth type of light, **THREE.SpotLight**. A spotlight is similar to a point light, except that instead of shining in all directions, a spotlight only produces a cone of light.
- The vertex of the cone is located at the position of the light.
- By default, the axis of the cone points from that location towards the origin. The constructor adds two parameters to those for a point light:

```
new THREE.SpotLight( color, intensity, cutoff, coneAngle, exponent )
```

A number between 0 and $\text{Math.PI}/2$ that determines the size of the cone of light. It is the angle between the axis of the cone and the side of the cone. Default value is $\text{Math.PI}/3$.

A non-negative number that determines how fast the intensity of the light decreases as we move from the axis of the cone toward the side. The default value, 10, gives a reasonable result. An exponent of zero gives no falloff at all, so that objects at all distances from the axis are evenly illuminated.

Lights

- An object `spot` of type `SpotLight` has a property named `spot.target`.
- The `target` is a scene graph node. The cone of light from the spotlight is pointed in the direction from spotlight's position towards the target's position.
- When a spotlight is first created, its target is a new, empty `Object3D`, with position at (0,0,0). However, we can set the target to be any object in the scene graph, which will make the spotlight shine towards that object.
- For three.js to calculate the spotlight direction, a target whose position is anything other than the origin must actually be a node in the scene graph. For example, suppose we want a spotlight located at the point (0,0,5) and pointed towards the point (2,2,0):

```
spotlight = new THREE.SpotLight();  
spotlight.position.set(0,0,5);  
spotlight.target.position.set(2,2,0);  
scene.add(spotlight);  
scene.add(spotlight.target);
```

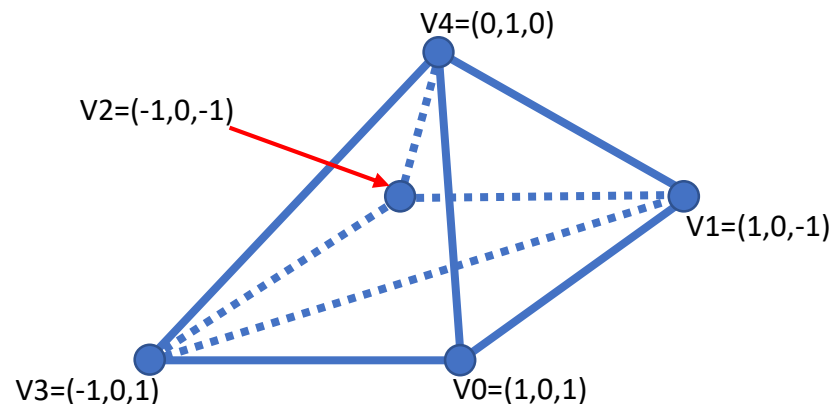


DOUGLAS COLLEGE

Polygonal Meshes and IFS

Polygonal Meshes

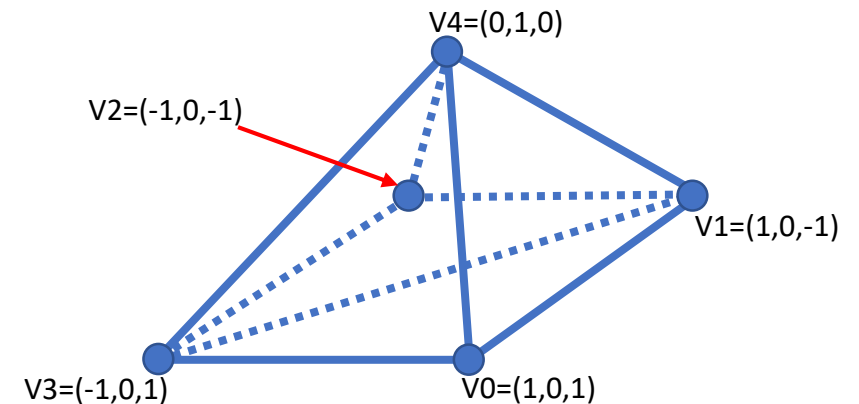
- A mesh in three.js is what we called a polygonal mesh in OpenGL, although in a three.js mesh, all of the polygons must be triangles.
- There are two ways to draw polygonal meshes in WebGL.
 - Use the function `glDrawArrays()`, which requires just a list of vertices.
 - Use the representation that we called an indexed face set (IFS), which is drawn using the function `glDrawElements()`. In addition to a list of vertices, an IFS uses a list of face indices to specify the triangles.
- We will examine both methods, using this pyramid as an example:



Polygonal Meshes

- The vertices are numbered from 0 to 4.
- A triangular face can be specified by the three numbers that give the vertex numbers of the vertices of that triangle.
- The vertices of a triangle should be specified in counterclockwise order when viewed from the front, that is, from outside the pyramid.

VERTEX COORDINATES	FACE INDICES
Vertex 0: 1, 0, 1	Face 1: 3, 2, 1
Vertex 1: 1, 0, -1	Face 2: 3, 1, 0
Vertex 2: -1, 0, -1	Face 3: 3, 0, 4
Vertex 3: -1, 0, 1	Face 4: 0, 1, 4
Vertex 4: 0, 1, 0	Face 5: 1, 2, 4
	Face 6: 2, 3, 4



Polygonal Meshes

- A three.js mesh object requires a geometry and a material. The geometry is an object of type `THREE.BufferGeometry`, which has a "position" attribute that holds the coordinates of the vertices that are used in the mesh.
- The attribute uses a typed array that holds the coordinates of the vertices of the triangles that make up the mesh. Geometry for the pyramid can be created using:

```
let pyramidVertices = new Float32Array( [
    // Data for the pyramidGeom "position" attribute.
    // Contains the x,y,z coordinates for the vertices.
    // Each group of three numbers is a vertex;
    // each group of three vertices is one face.
    -1,0,1,  -1,0,-1,  1,0,-1, // First triangle in the base.
    -1,0,1,   1,0,-1,  1,0,1,  // Second triangle in the base.
    -1,0,1,   1,0,1,   0,1,0,  // Front face.
    1,0,1,    1,0,-1,  0,1,0,  // Right face.
    1,0,-1,  -1,0,-1,  0,1,0,  // Back face.
    -1,0,-1, -1,0,1,   0,1,0   // Left face.
] );
let pyramidGeom = new THREE.BufferGeometry();
pyramidGeom.setAttribute("position",
    new THREE.BufferAttribute(pyramidVertices,3) );
```

Polygonal Meshes

- When this geometry is used with a Lambert or Phong material, normal vectors are required for the vertices.
- If the geometry has no normal vectors, Lambert and Phong materials will appear black.
- The normal vectors for a mesh have to be stored in another attribute of the **BufferedGeometry**.
- The name of the attribute is "**normal**", and it holds a normal vector for each vertex in the "**position**" attribute.
- It could be created in the same way that the "**position**" attribute is created, but a **BufferedGeometry** object includes a method for calculating normal vectors. For the **pyramidGeom**, we can simply call **pyramidGeom.computeVertexNormals()**;
- For a basic polygonal mesh, this will create normal vectors that are perpendicular to the faces.
- This is **not** appropriate if the polygonal mesh is being used to approximate a smooth surface.

Polygonal Meshes

- Once we have the geometry for our pyramid, we can use it in a three.js mesh object by combining it with, say, a yellow Lambert material:

```
pyramid = new THREE.Mesh(  
    pyramidGeom,  
    new THREE.MeshLambertMaterial({ color: "yellow" })  
);
```

- It is also possible to use different materials on different faces of a mesh.
- For that, the vertices in the geometry must be divided into groups.
- The `addGroup()` method in the `BufferedGeometry` class is used to create the groups.
- The vertices in the geometry are numbered 0, 1, 2, ..., according their sequence in the `"position"` attribute.
- The `addGroup()` method takes three parameters: the number of the first vertex in the group, the number of vertices in the group, and a material index. The material index is an integer that determines which material will be applied to the group.

Polygonal Meshes

- When using groups, it is important to put all of the vertices into groups. Here is how groups can be created for the pyramid:

```
pyramidGeom.addGroup(0,6,0); // The base (2 triangles)
pyramidGeom.addGroup(6,3,1); // Front face.
pyramidGeom.addGroup(9,3,2); // Right face.
pyramidGeom.addGroup(12,3,3); // Back face.
pyramidGeom.addGroup(15,3,4); // Left face.
```

- To apply different materials to different groups, the materials should be put into an array. The material index of a group is an index into that array:

```
pyramidMaterialArray= [
    // Array of materials, for use as pyramids's
    material.
    new THREE.MeshLambertMaterial( { color: 0xffffffff } ),
    new THREE.MeshLambertMaterial( { color: 0x99ffff } ),
    new THREE.MeshLambertMaterial( { color: 0xff99ff } ),
    new THREE.MeshLambertMaterial( { color: 0xffff99 } ),
    new THREE.MeshLambertMaterial( { color: 0xff9999 } )
];
```

This array can be passed as the second parameter to the **THREE.Mesh** constructor, where a single material would ordinarily be used.

```
pyramid = new THREE.Mesh( pyramidGeom, pyramidMaterialArray );
```

In-Class Example

The sample program uses the code from the preceding slides to create a pyramid, and it displays both the pyramid and a cube, using multiple materials on each object.

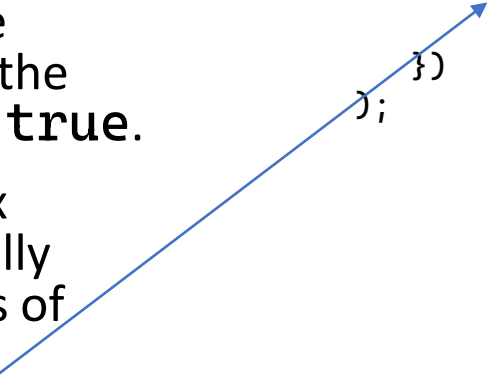
[Three.js Using different materials on different faces](#)



Polygonal Meshes

- There is another way to assign different colors to different vertices.
- A **BufferedGeometry** can have an attribute named "color" that specifies a color for each vertex.
- The "color" attribute uses an array containing a set of three RGB component values for each vertex.
- The vertex colors are ignored by default.
- To use them, the geometry must be combined with a material in which the **vertexColors** property is set to **true**.
- The color components of the vertex colors from the geometry are actually multiplied by the color components of the color in the Lambert material.

```
pyramidGeom.setAttribute(
    "color",
    new THREE.BufferAttribute( new Float32Array([
        1,1,1, 1,1,1, 1,1,1, // Base white
        1,1,1, 1,1,1, 1,1,1,
        1,0,0, 1,0,0, 1,0,0, // Front red
        0,1,0, 0,1,0, 0,1,0, // Right green
        0,0,1, 0,0,1, 0,0,1, // Back blue
        1,1,0, 1,1,0, 1,1,0 // Left yellow
    ]), 3)
);
pyramid = new THREE.Mesh(
    pyramidGeom,
    new THREE.MeshLambertMaterial({
        color: "white",
        vertexColors: true
    })
);
```



IFS Representation

- The `glDrawElements()` function uses the indexed face set pattern, which requires an array of face indices to specify the vertices for the faces of the mesh. In that array, a vertex is specified by a single number, rather than repeating all of the coordinates and other data for that vertex.
- Note that a given vertex number refers to all of the data for that vertex: vertex coordinates, normal vector, vertex color, and any other data that are provided in attributes of the geometry.
- Suppose that two faces share a vertex. If that vertex has a different normal vector, or a different value for some other attribute, in the two faces, then that vector will need to occur twice in the attribute arrays.
- The two occurrences can be combined only if the vertex has identical properties in the two faces.
- The IFS representation is most suitable for a polygonal mesh that is being used as an approximation for a smooth surface, since in that case a vertex has the same normal vector for all of the vertices in which it occurs.
- It can also be appropriate for an object that uses a `MeshBasicMaterial`, since normal vectors are not used with that type of material.



IFS Representation

- To use the IFS pattern with a **BufferedGeometry**, we need to provide a face index array for the geometry.
- The array is specified by the geometry's **setIndex()** method. The parameter can be an ordinary JavaScript array of integers.
- For our pyramid example the "**position**" attribute of the geometry would contain each vertex just once, and the face index array would refer to a vertex by its position in that list of vertices.
- The **computeVertexNormals()** method can still be used for a **BufferedGeometry** that has an index array.
- To compute a normal vector for a vertex, it finds all of the faces in which that vertex occurs. For each of those faces, it computes a vector perpendicular to the face. Then it averages those vectors to get the vertex normal.



```
pyramidVertices = new Float32Array( [
    1, 0, 1,  // vertex number 0
    1, 0, -1, // vertex number 1
    -1, 0, -1, // vertex number 2
    -1, 0, 1, // vertex number 3
    0, 1, 0    // vertex number 4
] );

pyramidFaceIndexArray = [
    3, 2, 1,  // First triangle in the base.
    3, 1, 0,  // Second Triangle in the base.
    3, 0, 4,  // Front face.
    0, 1, 4,  // Right face.
    1, 2, 4,  // Back face.
    2, 3, 4   // Left face.
];

pyramidGeom = new THREE.BufferGeometry();
pyramidGeom.setAttribute("position",
    new
    THREE.BufferAttribute(pyramidVertices,3) );
pyramidGeom.setIndex( pyramidFaceIndexArray );
```

In-Class Example

The sample program uses the code from the preceding slides to create a pyramid, and it displays both the pyramid and a cube, using multiple materials on each object.

[Three.js Using different materials on different faces](#)



Curves and Surfaces

Parametric Surfaces

- THREE.js has support for working with curves and surfaces that are defined mathematically.
- **Parametric surfaces** are the easiest to work with. A parametric surface is defined by a mathematical function of two parameters (u,v) , where u and v are numbers, and each value of the function is a point in space.
- The surface consists of all the points that are values of the function for u and v in some specified ranges.
- For three.js, the function is a regular JavaScript function that takes three parameters: u , v , and an object of type `THREE.Vector3`.
- The function must modify the vector to represent the point in space that corresponds to the values of the u and v parameters.
- A parametric surface geometry is created by calling the function at a grid of (u,v) points. This gives a collection of points on the surface, which are then connected to give a polygonal approximation of the surface.

Parametric Surfaces

- The geometry is created by a constructor:
`new THREE.ParametricGeometry( func, slices, stacks )`
- where `func` is the JavaScript function that defines the surface, and `slices` and `stacks` determine the number of points in the grid; `slices` gives the number of subdivisions of the interval from 0 to 1 in the `u` direction, and `stacks`, in the `v` direction.
- Once we have the geometry, we can use it to make a mesh in the usual way:

```
function surfaceFunction( u, v, vector ) {
    let x,y,z; // Coordinates for a point on the surface,
               // calculated from u,v, where u and v
               // range from 0.0 to 1.0.
    x = 20 * (u - 0.5); // x and z range from -10 to 10
    z = 20 * (v - 0.5);
    y = 2*(Math.sin(x/2) * Math.cos(z));
    vector.set( x, y, z );
}
let surfaceGeometry = new
THREE.ParametricGeometry(surfaceFunction, 64, 64);
let surface = new THREE.Mesh( surfaceGeometry, material );
```

Curves

- The class **THREE . Curve** represents the abstract idea of a parametric curve in two or three dimensions. (It does not represent a three.js geometry.)
- A parametric curve is defined by a function of one numeric variable t . The value returned by the function is of type **THREE . Vector2** for a 2D curve or **THREE . Vector3** for a 3D curve.
- For an object, **curve** of type **THREE . Curve**, the method **curve . getPoint(t)** should return the point on the curve corresponding to the value of the parameter t .
- The curve consists of points generated by this function for values of t ranging from 0.0 to 1.0.
- However, in the **Curve** class itself, **getPoint()** is undefined.
- To get an actual curve, we have to define it ourselves.

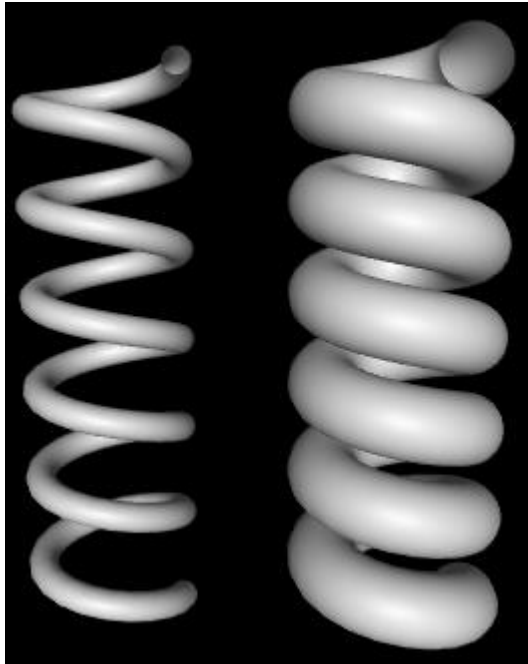
Curves

- ```
let helix = new THREE.Curve();
helix.getPoint = function(t) {
 let s = (t - 0.5) * 12*Math.PI;
 // As t ranges from 0 to 1, s ranges from -6*PI to 6*PI
 return new THREE.Vector3(
 5*Math.cos(s),
 s,
 5*Math.sin(s)
);
}
```
- 1.0.
- However, in the `Curve` class itself, `getPoint()` is undefined.
- To get an actual curve, we have to define it ourselves.

## Surfaces from 2D Curves

- Once `getPoint()` is defined, we have a usable curve.
- One thing to do with it is to create a tube geometry, which defines a surface that is a tube with a circular cross-section and with the curve running along the center of the tube.

```
tubeGeometry1 = new THREE.TubeGeometry(helix, 128, 2.5, 32);
```



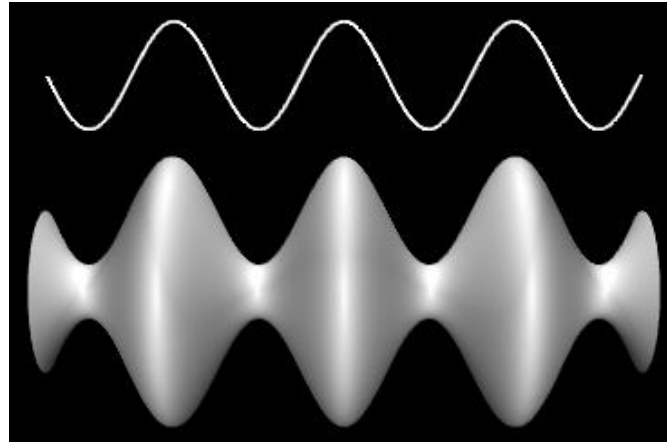
The number of subdivisions of the surface along the length of the curve.

the radius of the circular cross-section.

The number of subdivisions around the circumference of the cross-section.

## Surfaces from 2D Curves

- There are several ways to make a surface from a 2D curve.
- One way is to revolve the curve about a line, generating a surface of rotation.
- The surface consists of all the points that the curve passes through as it revolves. This is called **lathing**.



- The surface is created in three.js using a `THREE.LatheGeometry` object.
- A `LatheGeometry` is constructed not from a curve but from an array of points that lie on the curve.

## Surfaces from 2D Curves

- The points are objects of type **Vector2**, and the curve lies in the xy-plane.
- The surface is generated by rotating the curve about the y-axis. The **LatheGeometry** constructor takes the form:

```
new THREE.LatheGeometry(points, slices)
```

The array of **Vector2**.

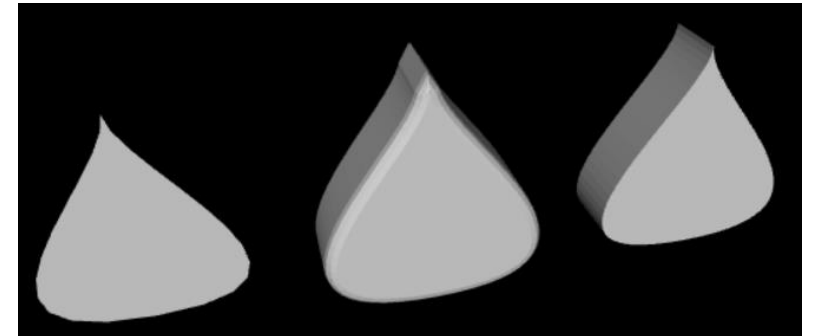
The number of subdivisions of the surface along the circle generated when a point is rotated about the axis.

- In the next [example](#), we create the array of points from an object, **cosine**, of type **Curve** by calling **cosine.getPoints(128)**.
- This function creates an array of 128 points on the curve, using values of the parameter that range from 0.0 to 1.0.

## Surfaces from 2D Curves

- We can also simply fill in the inside of the curve, giving a 2D filled shape by using an object of type `THREE.Shape`, which is a subclass of `THREE.Curve`.
- A Shape can be defined in the same way as a path in the 2D Canvas API.
  - An object shape of type `THREE.Shape` has methods `shape.moveTo`, `shape.lineTo`, `shape.quadraticCurveTo` and `shape.bezierCurveTo` that can be used to define the path.

```
let path = new THREE.Shape();
path.moveTo(0,10);
path.bezierCurveTo(0,5, 20,-10, 0,-10);
path.bezierCurveTo(-20,-10, 0,5, 0,10);
let shapeGeom = new THREE.ShapeGeometry(path);
```



- We can also **extrude** a filled shape (i.e., a filled 2D shape is moved along a path in 3D).
- For more details on extrusion, see the [documentation](#) for `THREE.ExtrudeGeometry` and the source code for the previous [example](#).





DOUGLAS COLLEGE

# Textures

## Textures

- In three.js, an image texture is represented by an object of type **THREE.Texture**.
- Since we are talking about web pages, the image for a three.js texture is generally loaded from a web address.
- Image textures are usually created using the load function in an object of type **THREE.TextureLoader**. The function takes a URL as parameter and returns a Texture object:

```
let loader = new THREE.TextureLoader();
let texture = loader.load(imageURL);
```
- A texture in three.js is considered to be part of a material. To apply a texture to a mesh, qw assign the Texture object to the map property of the mesh material that is used on the mesh:

```
material.map = texture;
```
- All three types of mesh material (Basic, Lambert, and Phong) can use a texture. In general, the material base color will be white, since the material color will be multiplied by colors from the texture.

## Textures

- Using a texture on an object before the image has finished loading does not cause an error, but the object will be rendered as completely black.
- Once the image has been loaded, the scene has to be rendered again to show the image texture.
- If an animation is running, this will happen automatically; the image will appear in the first frame after it has finished loading. But if there is no animation, we need a way to render the scene once the image has loaded.

- Thankfully, the `load()` function in a `TextureLoader` has several optional parameters:

```
loader.load(imageURL, onLoad, undefined, onError);
```

Parameter is no longer used in modern versions of WebGL.

- The `onLoad` and `onError` parameters are callback functions. The `onLoad` function, if defined, will be called once the image has been successfully loaded. The `onError` function will be called if the attempt to load the image fails.

```
texture = new THREE.TextureLoader().load("brick.png", render);
```

## Textures

- When you change the value of `material.map`, set `material.needsUpdate = true`; to ensure that the change will take effect when the object is redrawn.
- For a `Texture` object `tex`, the properties `tex.wrapS` and `tex.wrapT` control how `s` and `t` texture coordinates outside the range 0 to 1 are treated. **The default is "clamp to edge."**
- We most commonly want to make the texture repeat in both directions by setting the property values to `THREE.RepeatWrapping`:

```
tex.wrapS = THREE.RepeatWrapping;
tex.wrapT = THREE.RepeatWrapping;
```

- Three.js also offers an interesting variation called "mirrored repeat" in which every other copy of the repeated image is flipped.

```
tex.wrapS = THREE.MirroredRepeatWrapping;
tex.wrapT = THREE.MirroredRepeatWrapping;
```

## Textures

- The texture properties **repeat**, **offset**, and **rotation** control the scaling, translation, and rotation that are applied to the texture as texture transformations.
- The values of **repeat** and **offset** are of type **THREE.Vector2**, so that each property has an x and a y component.
- The **rotation** is a number, measured in **radians**, giving the rotation of the texture about the point (0,0). (The center of rotation is actually given by another property named **center**.)
- For a **Texture**, **tex**, the two components of **tex.offset** give the texture translation in the horizontal and vertical directions. To offset the texture by 0.5 horizontally:

```
tex.offset.x = 0.5;
// or
tex.offset.set(0.5, 0);
```

- A positive horizontal offset will move the texture to the left on the objects, because the offset is applied to the texture coordinates not to the texture image itself.

## Textures

- The components of the property `tex.repeat` give the texture scaling in the horizontal and vertical directions.
- This command will scale the texture coordinates by a factor of 2 horizontally and 3 vertically.

```
tex.repeat.set(2,3);
```

- 
- Again, the effect on the image is the inverse, so that the image is shrunk by a factor of 2 horizontally and 3 vertically.
- The result is that we get two copies of the image in the horizontal direction where you would have had one, and three vertically.
- See [this](#) example for an application of various texture images using THREE.js.

## Textures

- To apply a texture image to an object, WebGL needs texture coordinates for that object.
- When we build a mesh from scratch, we have to supply the texture coordinates as part of the mesh's geometry object.
- A **BufferedGeometry** object such as **pyramidGeom** in the example has an attribute named "uv" to hold texture coordinates.
- The name "uv" refers to the coordinates on an object that are mapped to the s and t coordinates in a texture. The texture coordinates for a surface are often referred to as "**uv coordinates**."
- The **BufferAttribute** for a "uv" attribute can be made from a typed array containing a pair of texture coordinates for each vertex.
- Our pyramid example has six triangular faces, with a total of 18 vertices. We need an array containing vertex coordinates for 18 vertices.
- The coordinates have to be chosen to map the image in a reasonable way onto the faces.

## Textures

- These coordinates map the entire texture image onto the square base of the pyramid, and they cut a triangle out of the image to apply to each of the sides.

```
let pyramidUVs = new Float32Array([
 0,0, 0,1, 1,1, // uv coords for first triangle in base.
 0,0, 1,1, 1,0, // uv coords for second triangle in base.
 0,0, 1,0, 0.5,1, // uv coords for front face.
 1,0, 0,0, 0.5,1, // uv coords for right face.
 0,0, 1,0, 0.5,1, // uv coords for back face.
 1,0, 0,0, 0.5,1 // uv coords for left face.
]);
pyramidGeom.setAttribute("uv",
 new THREE.BufferAttribute(pyramidUVs,2));
```

- [Here](#) is an example of how this can be done in practice.



## References

- Interactive Computer Graphics, a Top-Down Approach with WebGL, 7<sup>th</sup> Edition – Edward Angel and Dave Shreiner, Pearson.
- David J. Eck, Introduction to Computer Graphics, Version 1.3, August 2021. Notes and demos from this source are licensed under a [Creative Commons Attribution-Noncommercial-ShareAlike 4.0 License](#).



**Thank you.**  
**DOUGLAS**COLLEGE

**DOUGLAS**COLLEGE

# CMPT 2361

## Computer Graphics

Shahriar Khosravi, Ph.D.

Lecture 8

## Plan for this Lecture

- Introduction to WebGL
- The Programmable Pipeline
- The WebGL Graphics Context
- The Shader Program
- Data Flow in the Pipeline
- Values for Uniforms
- Values for Attributes
- Vertex Array Objects
- Instanced Drawing



DOUGLAS COLLEGE

# Introduction to WebGL

## Introduction to WebGL

- There are two versions of WebGL, both of them still in use.
- WebGL 1.0 is based on OpenGL ES 2.0, a version of OpenGL designed for use on embedded systems such as smart phones and tablets.
- OpenGL ES 1.0, the original OpenGL for embedded systems, was very similar to OpenGL 1.1. OpenGL ES 2.0 introduced major changes.
- It is a smaller, simpler API that puts more responsibility on the programmer.
  - For example, functions for working with transformations, such as `glRotatef()` and `glPushMatrix()`, were eliminated from the API, making the programmer responsible for keeping track of transformations.
- WebGL does not use `glBegin()/glEnd()` to generate geometry, and it doesn't use function such as `glColor*()` or `glNormal*()` to specify attributes of vertices.



## Introduction to WebGL

- WebGL 1.0 is supported in almost every web browser.
- WebGL 2.0, which is based on OpenGL ES 3.0, is already supported in many devices, but is still not as widely supported as the 1.0 version.
- Most programs written for WebGL 1.0 will also work under WebGL 2.0, so almost everything we learn about the 1.0 version will carry over to the newer version.
- Here, we will concentrate on WebGL 1.0, but we will also cover some of the new features of WebGL 2.0.
- There are two sides to any WebGL program. Part of the program is written in JavaScript, the programming language for the web. The second part is written in GLSL, a language for writing "shader" programs that run on the GPU.
- WebGL 1.0 uses GLSL ES 1.00 (the OpenGL Shader Language for Embedded Systems, version 1.00). WebGL 2.0 can use shader programs written in GLSL ES 1.00, but it can also use GLSL ES 3.00, which has some significant differences as well as new features.



DOUGLAS COLLEGE

# The Programmable Pipeline



# The Programmable Pipeline

- OpenGL 1.1 uses a **fixed-function pipeline** for graphics processing:
  - Data is provided by a program and passes through a series of processing stages that ultimately produce the pixel colors in the final image.
  - The program can enable and disable some of the steps in the process, such as the depth test and lighting calculations, but there is no way for it to change what happens at each stage. The functionality is fixed.
- OpenGL 2.0 introduced a **programmable pipeline**.
  - It became possible for the programmer to **replace certain stages in the pipeline with their own programs**.
  - This gives the programmer complete control over what happens at that stage.
- In OpenGL 2.0, the programmability was optional; the complete fixed-function pipeline was still available for programs that didn't need the flexibility of programmability.
- **WebGL uses a programmable pipeline, and it is mandatory.** There is no way to use WebGL without writing programs to implement part of the graphics processing pipeline.

## Vertex and Fragment Shaders

- The programs that are written as part of the pipeline are called **shaders**.
- For WebGL, we need to write a **vertex shader**, which is called once for each vertex in a primitive, and a **fragment shader**, which is called once for each pixel in the primitive.
- Aside from these two programmable stages, the WebGL pipeline also contains several stages from the original fixed-function pipeline.
  - For example, the depth test is still part of the fixed functionality, and it can be enabled or disabled in WebGL in the same way as in OpenGL 1.1.
- We first need to learn the basic structure of a WebGL program and how data flows from the JavaScript side of the program into the graphics pipeline and through the vertex and fragment shaders.



DOUGLAS COLLEGE

# The WebGL Graphics Context

## The WebGL Graphics Context

- To use WebGL, we need a WebGL **graphics context**. The graphics context is a JavaScript object whose methods implement the JavaScript side of the WebGL API.
- WebGL draws its images in an HTML canvas, the same kind of `<canvas>` element we have seen before.
- A graphics context is associated with a particular canvas. A graphics context for WebGL 1.0 can be obtained by calling the function `canvas.getContext("webgl")`, where `canvas` is a DOM object representing the canvas.
- For WebGL 2.0, we would simply use `canvas.getContext("webgl2")` instead.
- The return value of `getContext()` will be `null` if the context cannot be created.

```
canvas = document.getElementById("webglcanvas");
gl = canvas.getContext("webgl"); // or maybe canvas.getContext("webgl2")
if (! gl) {
 throw new Error("WebGL not supported; can't create graphics context.");
}
```

## The WebGL Graphics Context

```
canvas = document.getElementById("webglcanvas");
gl = canvas.getContext("webgl"); // or maybe canvas.getContext("webgl2")
if (! gl) {
 throw new Error("WebGL not supported; can't create graphics context.");
}
```

- Here, the first line gets a reference to the HTML canvas that WebGL will use throughout the program for drawing.
- The name `gl` for the variable is up to you.
- This code assumes that the HTML source for the web page includes a canvas element with `id="webglcanvas"`, such as:

```
<canvas width="800" height="600" id="webglcanvas"></canvas>
```

## The WebGL Graphics Context

```
canvas = document.getElementById("webglcanvas");
gl = canvas.getContext("webgl"); // or maybe canvas.getContext("webgl2")
if (! gl) {
 throw new Error("WebGL not supported; can't create graphics context.");
}
```

- In the second line of the above code, `canvas.getContext("webgl")` will return `null` if the web browser does not support "webgl" as a parameter to `getContext()`.
- Since `null` is considered to be `false` in JavaScript when used in a Boolean context, the third line tests whether the return value is `null`.
- In that case, the code throws an error, which can be handled by showing an error message to the user.
- The code will throw an exception if the browser has no support at all for `<canvas>`.

## The WebGL Graphics Context

```
canvas = document.getElementById("webglcanvas");
gl = canvas.getContext("webgl"); // or maybe canvas.getContext("webgl2")
if (! gl) {
 throw new Error("WebGL not supported; can't create graphics context.");
}
```

- In the second line of the above code, `canvas.getContext("webgl")` will return `null` if the web browser does not support "webgl" as a parameter to `getContext()`.
- Since `null` is considered to be `false` in JavaScript when used in a Boolean context, the third line tests whether the return value is `null`.
- In that case, the code throws an error, which can be handled by showing an error message to the user.
- The code will throw an exception if the browser has no support at all for `<canvas>`.

## The WebGL Graphics Context

- To use WebGL, we need a WebGL **graphics context**. The graphics context is a JavaScript object whose methods implement the JavaScript side of the WebGL API.
- WebGL draws its images in an HTML canvas, the same kind of `<canvas>` element we have seen before.
- A graphics context is associated with a particular canvas. A graphics context for WebGL 1.0 can be obtained by calling the function `canvas.getContext("webgl")`, where `canvas` is a DOM object representing the canvas.
- For WebGL 2.0, we would simply use `canvas.getContext("webgl2")` instead.
- The return value of `getContext()` will be `null` if the context cannot be created.,



# The WebGL Graphics Context

- Our WebGL programs often use an initialization function of the form:

```
function init() {
 try {
 canvas = document.getElementById("webglcanvas");
 gl = canvas.getContext("webgl"); // or "webgl2"
 if (! gl) {
 throw new Error("WebGL not supported.");
 }
 }
 catch (e) {
 .
 . // report the error
 .
 return;
 }
 initGL(); // a function that initializes the WebGL graphics context
 .
 . // other JavaScript initialization
 .
}
```

## The WebGL Graphics Context

- Here, `canvas` and `gl` are global variables.
- Function `initGL()` is a function defined elsewhere in the script that initializes the graphics context, including creating and installing the shader programs.
- The `initGL()` function should be called when the page is loaded. This can be arranged, for example, by assigning `"window.onload = init;"` in the script.
- Once the graphics context `gl`, has been created, it can be used to call functions in the WebGL API. For example, the command for enabling the depth test, which was written as `glEnable(GL_DEPTH_TEST)` in OpenGL, becomes  
`gl.enable( gl.DEPTH_TEST );`
- Note that both functions and constants in the API are referenced through the graphics context.



# The Shader Program

## The Shader Program

- Drawing with WebGL requires a shader program, which consists of a vertex shader and a fragment shader.
- Shaders are written in some version of the GLSL programming language. WebGL 1.0 used GLSL ES 1.00, while WebGL 2.0 can use either GLSL ES 1.00 or GLSL ES 3.00. The discussion here is about GLSL ES 1.00.
- GLSL is based on the C programming language.
- The vertex shader and fragment shader are separate programs, each with its own `main()` function. The two shaders are compiled separately and then "**linked**" to produce a complete shader program.
- The JavaScript API for WebGL includes functions for compiling the shaders and then linking them. To use the functions, the source code for the shaders must be JavaScript strings.

## The Shader Program

- It takes three steps to create the vertex shader:

```
let vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertexShaderSource);
gl.compileShader(vertexShader);
```

- The functions that are used here are part of the WebGL graphics context `gl`, and the parameter `vertexShaderSource` is the string that contains the source code for the shader.
- Errors in the source code will cause the compilation to fail silently. We need to check for compilation errors by calling the function:

```
gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS)
```

- This returns a Boolean value to indicate whether the compilation succeeded.
- In the event that an error occurred, we can retrieve an error message with:

```
gl.getShaderInfoLog(vertexShader)
```

- This returns a human-readable string containing the result of the compilation.

## The Shader Program

- The fragment shader can be created in a similar way. With both shaders in hand, we can create and link the program.
- The shaders need to be "**attached**" to a program object before linking. The code takes the form:

```
let prog = gl.createProgram();
gl.attachShader(prog, vertexShader);
gl.attachShader(prog, fragmentShader);
gl.linkProgram(prog);
```

- Even if the shaders have been successfully compiled, errors can occur when they are linked into a complete program.
- For example, the vertex and fragment shader can share certain kinds of variable. If the two programs declare such variables with the same name but with different types, an error will occur at link time.
- Checking for link errors is similar to checking for compilation errors in the shaders.

# The Shader Program

- The code for creating a shader program is trivial, so it is convenient to pack it into a reusable function.

```
function createProgram(gl, vertexShaderSource, fragmentShaderSource) {
 let vsh = gl.createShader(gl.VERTEX_SHADER);
 gl.shaderSource(vsh, vertexShaderSource);
 gl.compileShader(vsh);
 if (! gl.getShaderParameter(vsh, gl.COMPILE_STATUS)) {
 throw new Error("Error in vertex shader: " + gl.getShaderInfoLog(vsh));
 }
 let fsh = gl.createShader(gl.FRAGMENT_SHADER);
 gl.shaderSource(fsh, fragmentShaderSource);
 gl.compileShader(fsh);
 if (! gl.getShaderParameter(fsh, gl.COMPILE_STATUS)) {
 throw new Error("Error in fragment shader: " + gl.getShaderInfoLog(fsh));
 }
 let prog = gl.createProgram();
 gl.attachShader(prog, vsh);
 gl.attachShader(prog, fsh);
 gl.linkProgram(prog);
 if (! gl.getProgramParameter(prog, gl.LINK_STATUS)) {
 throw new Error("Link error in program: " + gl.getProgramInfoLog(prog));
 }
 return prog;
}
```

# The Shader Program

```
function createProgram(gl, vertexShaderSource, fragmentShaderSource) {
 let vsh = gl.createShader(gl.VERTEX_SHADER);
 gl.shaderSource(vsh, vertexShaderSource);
 gl.compileShader(vsh);
 if (! gl.getShaderParameter(vsh, gl.COMPILE_STATUS)) {
 throw new Error("Error in vertex shader: " + gl.getShaderInfoLog(vsh));
 }
 let fsh = gl.createShader(gl.FRAGMENT_SHADER);
 gl.shaderSource(fsh, fragmentShaderSource);
 gl.compileShader(fsh);
 if (! gl.getShaderParameter(fsh, gl.COMPILE_STATUS)) {
 throw new Error("Error in fragment shader: " + gl.getShaderInfoLog(fsh));
 }
 let prog = gl.createProgram();
 gl.attachShader(prog, vsh);
 gl.attachShader(prog, fsh);
 gl.linkProgram(prog);
 if (! gl.getProgramParameter(prog, gl.LINK_STATUS)) {
 throw new Error("Link error in program: " + gl.getProgramInfoLog(prog));
 }
 return prog;
}
```

- We also have to tell the WebGL context to use the program. If **prog** is a program identifier returned by this above function, then:

```
gl.useProgram(prog);
```



## The Shader Program

- It is also possible to create several shader programs.
- This allows us to switch from one program to another at any time by calling `gl.useProgram()`, even in the middle of rendering an image.
  - Three.js, for example, uses a different program for each type of Material.
- As a matter of good coding practice, we should create any shader programs that we need as part of initialization.
- Although `gl.useProgram()` is a fast operation, compiling and linking are slow, so it is better to avoid creating new programs while in the process of drawing an image.
- Shaders and programs that are no longer needed can be deleted to free up the resources they consume.
- Use functions `gl.deleteShader(shader)` and `gl.deleteProgram(program)` to do so.



DOUGLAS COLLEGE

## Data Flow in the Pipeline

## Data Flow in the Pipeline

- The WebGL graphics pipeline renders an image. The data that defines the image comes from JavaScript.
- As data passes through the pipeline, it is processed by the current vertex shader and fragment shader as well as by the fixed-function stages of the pipeline.
- The basic operation in WebGL is to draw a geometric **primitive**.
- WebGL uses just seven of the OpenGL primitives. The primitives for drawing quads and polygons have been removed.
- The remaining primitives draw points, line segments, and triangles.
- In WegGL, the seven types of primitive are identified by the constants `gl.POINTS`, `gl.LINES`, `gl.LINE_STRIP`, `gl.LINE_LOOP`, `gl.TRIANGLES`, `gl.TRIANGLE_STRIP`, and `gl.TRIANGLE_FAN`, where `gl` is a WebGL graphics context.

## Data Flow in the Pipeline

- When WebGL is used to draw a primitive, there are two general categories of data that can be provided for the primitive:
  - attribute variables (or just "**attributes**"); and
  - uniform variables (or just "**uniforms**").
- The difference between attributes and uniforms is that a uniform variable has a single value that is the same for the entire primitive, while the value of an attribute variable can be different for different vertices.
- A primitive is defined by its type and by a list of vertices.
- One attribute that should always be specified is the coordinates of the vertex. The vertex coordinates must be an attribute since each vertex in a primitive will have its own set of coordinates.
- Another possible attribute is color. We have seen that OpenGL allows us to specify a different color for each vertex of a primitive.
- We can do the same thing in WebGL, and in that case the color will be an attribute. On the other hand, maybe we want the entire primitive to have the same, "uniform" color; in that case, color can be a uniform variable.

## Data Flow in the Pipeline

- Other quantities that could be either attributes or uniforms, depending on our needs, include normal vectors and material properties.
- Texture coordinates, if they are used, are almost certain to be an attribute, since it doesn't make sense for all the vertices in a primitive to have the same texture coordinates.
- If a geometric transform is to be applied to the primitive, it would naturally be represented as a uniform variable.
- WebGL does not come with any predefined attributes, not even one for vertex coordinates.
- In the programmable pipeline, the attributes and uniforms that are used are entirely up to the programmer.
- As far as WebGL is concerned, attributes are just values that are passed into the vertex shader. Uniforms can be passed into the vertex shader, the fragment shader, or both.
- WebGL does not assign a meaning to the values. The meaning is entirely determined by what the shaders do with the values.
- The set of attributes and uniforms that are used in drawing a primitive is determined by the source code of the shaders that are in use when the primitive is drawn.

## Data Flow in the Pipeline

- To understand this, we need to look at what happens in the pipeline in a more detail.
- When drawing a primitive, the JavaScript program specifies values for any attributes and uniforms in the shader program.
- For each attribute, it will specify an array of values, one for each vertex. For each uniform, it will specify a single value.
- It must send these values to the GPU before drawing the primitive. The primitive can then be drawn by calling a single JavaScript function.
- At that point, the GPU takes over, and executes the shader programs. When drawing the primitive, the GPU calls the vertex shader once for each vertex.
- The attribute values for the vertex that is to be processed are passed as input into the vertex shader. Values of uniform variables are also passed to the vertex shader.
- Both attributes and uniforms are represented as global variables in the vertex shader program. Before calling the shader for a given vertex, the GPU sets the values of those variables appropriately for that specific vertex.

## Data Flow in the Pipeline

- As one of its outputs, the vertex shader must specify the coordinates of the vertex in the clip coordinate system.
- It does so by assigning a value to a special variable named `gl_Position`.
- The position is often computed by applying a transformation to the attribute that represents the coordinates in the object coordinate system, **but exactly how the position is computed is up to the programmer**.
- After the positions of all the vertices in the primitive have been computed, a fixed-function stage in the pipeline clips away the parts of the primitive whose coordinates are outside the range of valid clip coordinates (-1 to 1 along each coordinate axis).
- The primitive is then rasterized (which pixels lie inside the primitive).
- The GPU then calls the fragment shader once for each pixel that lies in the primitive. The fragment shader has access to uniform variables (**but not attributes**).
- It can also use a special variable named `gl_FragCoord` that contains the clip coordinates of the pixel.
- Pixel coordinates are computed by interpolating the values of `gl_Position` that were specified by the vertex shader.
- The interpolation is done by another fixed-function stage that comes between the vertex shader and the fragment shader.

## Data Flow in the Pipeline

- Other quantities besides coordinates can work in the same way (i.e., the vertex shader computes a value for the quantity at each vertex of a primitive).
- An interpolator takes the values at the vertices and computes a value for each pixel in the primitive.
- The value for a given pixel is then input into the fragment shader when the shader is called to process that pixel.
- For example, color in OpenGL follows this pattern:
  - The color of an interior pixel of a primitive is computed by interpolating the color at the vertices.
  - In GLSL, this pattern is implemented using **varying variables**.



## Data Flow in the Pipeline

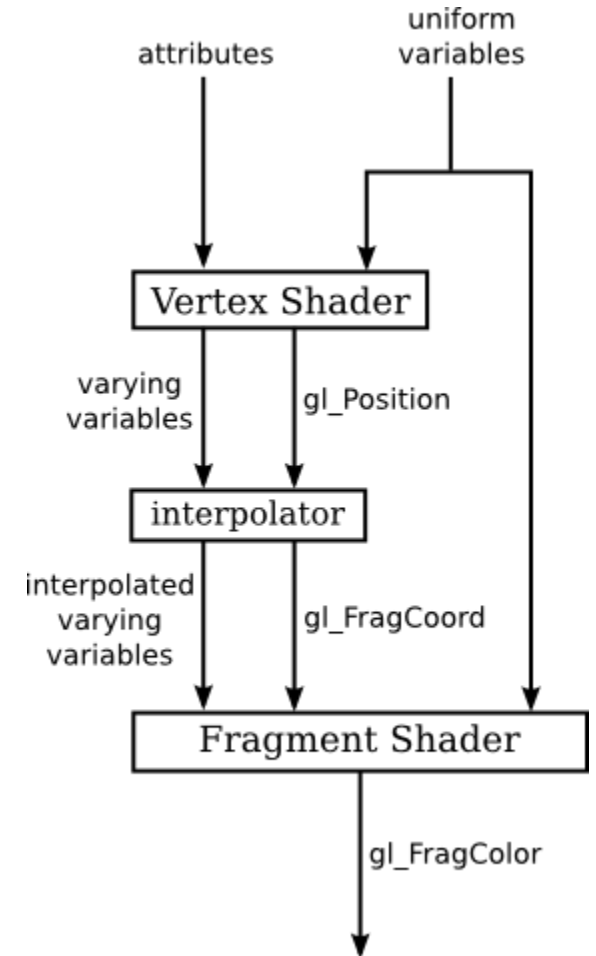
- A **varying** variable is declared **both in the vertex shader and in the fragment shader**.
- The vertex shader is responsible for assigning a value to the varying variable.
- Each vertex of a primitive can assign a different value to the variable.
- The interpolator takes all the values produced by executing the vertex shader for each vertex of the primitive, and it interpolates those values to produce a value for each pixel.
- When the fragment shader is executed for a given pixel, the value of the varying variable is the interpolated value for that pixel.
- The fragment shader can use the value in its own computations.
- **Varying** variables exist to communicate data from the vertex shader to the fragment shader.
- They are defined in the shader source code. They are **not** used or referred to in the JavaScript side of the API.
- It is entirely up to the programmer to decide what varying variables to define and what to do with them.

## Data Flow in the Pipeline

- After all that, the job of the fragment shader is simply to specify a color for the pixel.
- It does that by assigning a value to a special variable named `gl_FragColor`.
- That value will then be used in the remaining fixed-function stages of the pipeline.
- To summarize:
  1. The JavaScript side of the program sends values for attributes and uniform variables to the GPU and then issues a command to draw a primitive.
  2. The GPU executes the vertex shader once for each vertex. The vertex shader can use the values of attributes and uniforms. It assigns values to `gl_Position` and to any varying variables that exist in the shader.
  3. After clipping, rasterization, and interpolation, the GPU executes the fragment shader once for each pixel in the primitive.
  4. The fragment shader can use the values of varying variables, uniform variables, and `gl_FragCoord`. It computes a value for `gl_FragColor`.

## Data Flow in the Pipeline

- The following diagram is not complete.
- There are a few more special variables that we have not discussed.
- And there is the important question of how textures are used.



## Data Flow in the Pipeline

- For GLSL ES 3.00, the same diagram applies, except that there is no special variable `gl_FragColor`. Instead, the fragment shader must define its own output variable to represent the color.
- In GLSL ES 1.00, the words "attribute" and "varying" are used when declaring variables in the actual shader program source code.
- In source code for the 3.00 version, attribute variables become "in" variables, since they are inputs to the vertex shader, and varying variables become "out" variables in the vertex shader and "in" variables in the fragment shader.
- The variable `gl_FragColor` is replaced by an "out" variable in the fragment shader.
- The use of the terms "in" and "out" are more appropriate to systems with additional pipeline stages, where "out" variables from one stage can become "in" variables to the next stage.



DOUGLAS COLLEGE

## Values for Uniform Variables

## Values for Uniform Variables

- GLSL has some familiar basic data types: `float`, `int`, and `bool`. But it also has some new predefined data types to represent vectors and matrices.
  - For example, the data type `vec3` represents a vector in 3D.
  - The value of a `vec3` variable is a list of three floating-point numbers. Similarly, there are data types `vec2` and `vec4` to represent 2D and 4D vectors.
- Global variable declarations in a vertex shader can be marked as `attribute`, `varying`, or `uniform`.
- A variable declaration with none of these modifiers defines a variable that is **local** to the vertex shader.
- Global variables in a fragment can optionally be uniform or varying, or they can be declared without a modifier.
- **A varying variable should be declared in both shaders**, with the same name and type. This allows the GLSL compiler to determine what attribute, uniform, and varying variables are used in a shader program.

## Values for Uniform Variables

- The JavaScript side of the program needs a way to refer to particular attributes and uniform variables.
- The function `gl.getUniformLocation()` can be used to get a reference to a uniform variable in a shader program, where `gl` refers to the WebGL graphics context.
- It takes as parameters the identifier for the compiled program, which was returned by `gl.createProgram()`, and the name of the uniform variable in the shader source code.
- For example, if `prog` identifies a shader program that has a uniform variable named `color`, then the location of the color variable can be obtained with the JavaScript statement:

```
colorUniformLoc = gl.getUniformLocation(prog, "color");
```

- The location `colorUniformLoc` can then be used to set the value of the uniform variable. For example:

```
gl.uniform3f(colorUniformLoc, 1, 0, 0);
```

## Values for Uniform Variables

- The function `gl.uniform3f()` is one of a family of functions that can be referred to as a group as `gl.uniform*()`.
- This is similar to the family `glVertex*()` in OpenGL 1.1. The `*` represents a suffix that tells the number and type of values that are provided for the variable.
- `gl.uniform3f()` takes three floating point values, and it is used for setting the value of a uniform variable of type `vec3`. The number of values can be 1, 2, 3, or 4. The type can be "f" for floating point or "i" for integer.
- For a Boolean uniform, we can use `gl.uniform1i()` and pass 0 to represent false or 1 to represent true.
- If a "v" is added to the suffix, then the values are passed in an array. For example, `gl.uniform3fv( colorUniformLoc, [ 1, 0, 0 ] );`
- The value of a uniform variable can be set any time after the shader program has been compiled, and the value remains in effect until it is changed by another call to `gl.uniform*()`.



## Values for Uniform Variables

- If the string that is passed as the second parameter `gl.getUniformLocation()` is not the name of a uniform variable in the shader programs, then the return value is `null`.
- The return value can also be `null` if the uniform variable is declared in the shader source code but is not "**active**" in the program.
- A variable that is declared but not actually used is not active, and it does not get a location in the compiled program.
- So, be careful about commenting out part of a shader program for debugging purposes, and accidentally making a variable inactive.

## Values for Attributes

## Values for Attributes

- The situation is more complicated for attributes because an attribute can take a different value for each vertex in a primitive.
- The basic idea is that the complete set of data for the attribute is copied in a single operation from a JavaScript array into memory that is accessible to the GPU. Unfortunately, setting things up to make that operation possible is difficult.
- First, a regular JavaScript array is not suitable for this purpose. For efficiency, we need the data to be in a block of memory holding numerical values in successive memory locations, and regular JavaScript arrays don't have that form.
- To fix this problem, a new kind of array, called **typed arrays**, was introduced into JavaScript. We worked with typed arrays when working with three.js in Lab 5:
  - A typed array has a fixed length, which is assigned when it is created, and it can only hold numbers of a specified type.
  - There are different kinds of typed array for different kinds of numerical data. For now we will use Float32Array, which holds 32-bit floating point numbers.
  - Once you have a typed array, we can use it much like a regular array, but when we assign any value to an element of a Float32Array, the value is converted into a 32-bit floating point number.
  - If the value cannot be interpreted as a number, it will be converted to NaN, the "**not-a-number**" value.

## Values for Attributes

- Before data can be transferred from JavaScript into an attribute variable, it must be placed into a typed array.
- When possible, for efficiency, we should work with **typed arrays** directly, rather than working with regular JavaScript arrays and then copying the data into typed arrays.
- For use in WebGL, the attribute data must be transferred into a **VBO (vertex buffer object)**.
- A VBO is a block of memory that is accessible to the GPU. To use a VBO, we must first call the function `gl.createBuffer()` to create it. For example,

```
colorBuffer = gl.createBuffer();
```

- Before transferring data into the VBO, we must "bind" the VBO:

```
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
```

- The first parameter to `gl.bindBuffer()` is called the "**target**." It specifies how the VBO will be used. The target `gl.ARRAY_BUFFER` is used when the buffer is being used to store values for an attribute.
- Only one VBO at a time can be bound to a given target.

## Values for Attributes

- The function that transfers data into a VBO doesn't mention the VBO. Instead, it uses the VBO that is currently bound.
- To copy data into that buffer, we use `gl.bufferData()`. For example:  

```
gl.bufferData(gl.ARRAY_BUFFER, colorArray, gl.STATIC_DRAW);
```
- The first parameter is the target. The data is transferred into the VBO that is bound to that target.
- The second parameter is the typed array that holds the data on the JavaScript side. All the elements of the array are copied into the buffer, and the size of the array determines the size of the buffer.
- Note that this is a transfer of raw data in bytes. WebGL does not remember whether the data represents floating points or integers, etc.
- The third parameter is one of the constants `gl.STATIC_DRAW`, `gl.STREAM_DRAW`, or `gl.DYNAMIC_DRAW`.
- It is a hint to WebGL about how the data will be used, and it helps WebGL to manage the data in the most efficient way.

## Values for Attributes

- The value `gl.STATIC_DRAW` means that we intend to use the data many times without changing it.
- For example, if we will use the same data throughout the program, we can load it into a buffer once, during initialization, using `gl.STATIC_DRAW`.
- WebGL will store (or cache) the data on the graphics card itself where it can be accessed most quickly by the graphics hardware.
- `gl.STREAM_DRAW` is for data that will be used only once, or at most a few times. It can be "streamed" to the GPU when it is needed.
- The value `gl.DYNAMIC_DRAW` is somewhere between the other two values; it is meant for data that will be used multiple times, but with modifications.

## Values for Attributes

- Getting attribute data into VBOs is only part of the story. We also have to tell WebGL to use the VBO as the source of values for the attribute.
- To do so, we first need to know the location of the attribute in the shader program. We can determine that using `gl.getAttributeLocation()`. For example,

```
colorAttribLoc = gl.getAttributeLocation(prog, "a_color");
```

- Here, `prog` is the shader program and `"a_color"` is the name of the attribute variable in the vertex shader.
- This is analogous to `gl.getUniformLocation()`, except that the return value is an integer, and is -1 if the requested attribute does not exist or is not active.

## Values for Attributes

- Although an attribute usually takes different values at different vertices, it is possible to use the same value at every vertex. In fact, that is the default behavior.
- The single attribute value for all vertices can be set using the family of functions `gl.vertexAttrib*()`, which work similarly to `gl.uniform*()`.
- In the more usual case, where we want to take the values of an attribute from a VBO, we must enable the use of a VBO for that attribute:

```
gl.enableVertexAttribArray(colorAttribLoc);
```

- Here, the parameter is the location of the attribute in the shader program, as returned by a call to `gl.getAttribLocation()`.
- This command has nothing to do with any particular VBO. It just turns on the use of buffers for the specified attribute.
- Often, it is reasonable to call this method just once, during initialization.
- Use of data from the VBO can be also be turned off:

```
gl.disableVertexAttribArray(colorAttribLoc);
```



## Values for Attributes

- Finally, before we draw a primitive that uses the attribute data from a VBO, we have to tell WebGL which buffer contains the data and how the bits in that buffer are to be interpreted.
- This is done with `gl.vertexAttribPointer()`. The VBO must be bound to the `ARRAY_BUFFER` target when this function is called. For example,

```
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
gl.vertexAttribPointer(colorAttribLoc, 3, gl.FLOAT, false, 0, 0);
```

The attribute location.

number of values per vertex. For a `vec2`, it will be 2. For a `vec3`, it will be 3. For a `float`, it will be 1.

specifies the type of each value from `gl.FLOAT`, `gl.BYTE`, `gl.UNSIGNED_BYTE`, `gl.UNSIGNED_SHORT`, and `gl.SHORT` for integer values.

- Assuming that `colorBuffer` refers to the VBO and `colorAttribLoc` is the location of the attribute, this tells WebGL to take values for the attribute from that buffer.
- Often, we will call `gl.bindBuffer()` just before calling `gl.vertexAttribPointer()`, but that is not necessary if the desired buffer is already bound.

## Values for Attributes

- Finally, before we draw a primitive that uses the attribute data from a VBO, we have to tell WebGL which buffer contains the data and how the bits in that buffer are to be interpreted.
- This is done with `gl.vertexAttribPointer()`. The VBO must be bound to the `ARRAY_BUFFER` target when this function is called. For example,

```
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
gl.vertexAttribPointer(colorAttribLoc, 3, gl.FLOAT, false, 0, 0);
```

The attribute location.

number of values per vertex. For a `vec2`, it will be 2. For a `vec3`, it will be 3. For a `float`, it will be 1.

specifies the type of each value from `gl.FLOAT`, `gl.BYTE`, `gl.UNSIGNED_BYTE`, `gl.UNSIGNED_SHORT`, and `gl.SHORT` for integer values.

- Assuming that `colorBuffer` refers to the VBO and `colorAttribLoc` is the location of the attribute, this tells WebGL to take values for the attribute from that buffer.
- Often, we will call `gl.bindBuffer()` just before calling `gl.vertexAttribPointer()`, but that is not necessary if the desired buffer is already bound.

## Values for Attributes

- Finally, before we draw a primitive that uses the attribute data from a VBO, we have to tell WebGL which buffer contains the data and how the bits in that buffer are to be interpreted.
- This is done with `gl.vertexAttribPointer()`. The VBO must be bound to the `ARRAY_BUFFER` target when this function is called. For example,  

```
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
gl.vertexAttribPointer(colorAttribLoc, 3, gl.FLOAT, false, 0, 0);
```

The attribute location.

number of values per vertex. For a `vec2`, it will be 2. For a `vec3`, it will be 3. For a `float`, it will be 1.

specifies the type of each value from `gl.FLOAT`, `gl.BYTE`, `gl.UNSIGNED_BYTE`, `gl.UNSIGNED_SHORT`, and `gl.SHORT` for integer values.

- Note that in WebGL 1.0, all attributes are floating point values; if we provide integer values for an attribute, they will be converted to floating point. The third parameter value should match the data type in the buffer. For example, if the data came from a `Float32Array`, then the parameter should be `gl.FLOAT`.

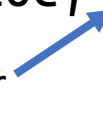
## Values for Attributes

- Finally, before we draw a primitive that uses the attribute data from a VBO, we have to tell WebGL which buffer contains the data and how the bits in that buffer are to be interpreted.
- This is done with `gl.vertexAttribPointer()`. The VBO must be bound to the `ARRAY_BUFFER` target when this function is called. For example,
 

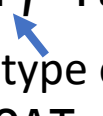
```
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
gl.vertexAttribPointer(colorAttribLoc, 3, gl.FLOAT, false, 0, 0);
```



The attribute location.



number of values per vertex. For a `vec2`, it will be 2. For a `vec3`, it will be 3. For a `float`, it will be 1.



specifies the type of each value from `gl.FLOAT`, `gl.BYTE`, `gl.UNSIGNED_BYTE`, `gl.UNSIGNED_SHORT`, and `gl.SHORT` for integer values.
- For the last three parameters, we will always use `false`, `0`, and `0`. These parameters add flexibility that we won't need; you can look them up in the documentation if you are interested

## Values for Attributes

- In WebGL 2.0, attribute variables can have type `int`.
- When `gl.vertexAttribPointer()` is used to configure an attribute, the values provided for the attribute will always be converted to floating point, so it is inappropriate for integer-valued attributes.
- For use with integer-valued attributes, WebGL 2.0 introduces a new function, `gl.vertexAttribIPointer()` that works correctly with integer data.
- Using a VBO to provide values for an attribute requires six separate commands, and that is in addition to generating the data and placing it in a typed array. Here is the full set of commands:

```
colorAttribLoc = gl.getAttribLocation(prog, "a_color");
colorBuffer = gl.createBuffer();
gl.enableVertexAttribArray(colorAttribLoc);
```

```
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
gl.vertexAttribPointer(colorAttribLoc, 3, gl.FLOAT, false, 0, 0);
gl.bufferData(gl.ARRAY_BUFFER, colorArray, gl.STATIC_DRAW);
```

## Values for Attributes

- These six commands will not always occur at the same point in the JavaScript code.
- The first three commands are often done as part of initialization.
- `gl.bufferData()` would be called whenever the data for the attribute needs to be changed; it might be used just once during initialization, or it might be used whenever the data needs to be modified.
- `gl.bindBuffer()` must be called before `gl.vertexAttribPointer()` or `gl.bufferData()`, since it establishes the VBO that is used by those two commands.
- Remember that all of this must be done for every attribute that is used in the shader program.

```
colorAttribLoc = gl.getAttribLocation(prog, "a_color");
colorBuffer = gl.createBuffer();
gl.enableVertexAttribArray(colorAttribLoc);
```

```
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
gl.vertexAttribPointer(colorAttribLoc, 3, gl.FLOAT, false, 0, 0);
gl.bufferData(gl.ARRAY_BUFFER, colorArray, gl.STATIC_DRAW);
```



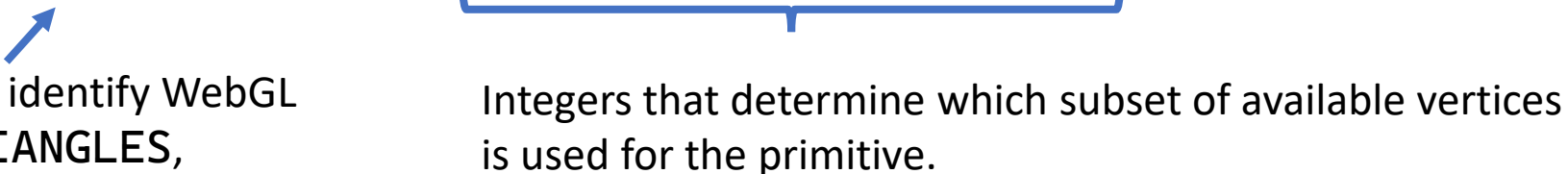
DOUGLAS COLLEGE

## Drawing a Primitive

## Drawing a Primitive

- After the shader program has been created and values have been set up for the uniform variables and attributes, it takes just one more command to draw a primitive. One way to do that is with the function `gl.drawArrays()`:

```
gl.drawArrays(primitiveType, startVertex, vertexCount);
```



One of the seven constants that identify WebGL primitive types, such as `gl.TRIANGLES`, `gl.LINE_LOOP`, and `gl.POINTS`.

Integers that determine which subset of available vertices is used for the primitive.

- Before calling `gl.drawArrays()`, we will have placed attribute values for some number of vertices into one or more VBOs.
- When the primitive is rendered, the attribute values for enabled attributes are pulled from the VBOs.
- The `startVertex` is the starting vertex number of the data within the VBOs, and `vertexCount` is the number of vertices in the primitive.



## Drawing a Primitive

- After the shader program has been created and values have been set up for the uniform variables and attributes, it takes just one more command to draw a primitive. One way to do that is with the function `gl.drawArrays()`:

```
gl.drawArrays(primitiveType, startVertex, vertexCount);
```

One of the seven constants that identify WebGL primitive types, such as `gl.TRIANGLES`, `gl.LINE_LOOP`, and `gl.POINTS`.

Integers that determine which subset of available vertices is used for the primitive.

- Often, `startVertex` is zero, and `vertexCount` is the total number of vertices for which data is available.
- For example, the command for drawing a single triangle might be

```
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

## Drawing a Primitive

- The use of the word "array" in `gl.drawArrays()` and `gl.ARRAY_BUFFER()` might be confusing, since the data is stored in vertex buffer objects rather than in JavaScript arrays.
- When `glDrawArrays()` was first introduced in OpenGL 1.1, it used ordinary arrays rather than VBOs.
- Starting with OpenGL 1.5, `glDrawArrays()` could be used either with ordinary arrays or VBOs.
- In WebGL, support for ordinary arrays was entirely dropped, and `gl.drawArrays()` can now only work with VBOs, even though the name still refers to arrays!
- We have also seen `glDrawElements()` in OpenGL, which can be used for drawing indexed face sets.
- A `gl.drawElements()` function is also available in WebGL.

## Drawing a Primitive

- With `gl.drawElements()`, attribute data is not used in the order in which it occurs in the VBOs. Instead, there is a separate list of indices that determines the order in which the data is accessed.
- To use `gl.drawElements()`, an extra VBO is required to hold the list of indices.
- When used for this purpose, the VBO must be bound to the target `gl.ELEMENT_ARRAY_BUFFER` rather than `gl.ARRAY_BUFFER`.
- The VBO will hold integer values, which can be of type `gl.UNSIGNED_BYTE` or `gl.UNSIGNED_SHORT` (or, for WebGL 2.0, `gl.UNSIGNED_INT`).
- The values can be loaded from a JavaScript typed array of type `Uint8Array`, for `gl.UNSIGNED_BYTE`, or `Uint16Array`, for `gl.UNSIGNED_SHORT`.
- Creating the VBO and filling it with data is again a multistep process.

## Drawing a Primitive

- For example:

```
elementBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, elementBuffer);
let data = new Uint8Array([2,0,3, 2,1,3, 1,4,3]);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, data, gl.STREAM_DRAW);
```

- Assuming that the attribute data has also been loaded into VBOs, `gl.drawElements()` can then be used to draw the primitive.
- A call to `gl.drawElements()` takes the form:

```
gl.drawElements(primitiveType, count, dataType, startByte);
```

A primitive type such  
as `gl.TRIANGLE_FAN`.

The number of vertices  
in the primitive.

The type of data that was  
loaded into the VBO; it will be  
either `gl.UNSIGNED_SHORT`  
or `gl.UNSIGNED_BYTE`.

The starting point in  
the VBO of the data  
for the primitive; it  
is usually zero.

- The VBO that contains the vertex indices must be bound to the `ELEMENT_ARRAY_BUFFER` target when this function is called.

## Drawing a Primitive

- Here is a typical example:

```
gl.drawElements(gl.TRIANGLES, 9, gl.UNSIGNED_BYTE, 0);
```



DOUGLAS COLLEGE

# Vertex Array Objects

## Vertex Array Objects

- The large number of functions needed to work with attributes can seem excessive.
- The situation is worse in a program that draws several different objects.
- Each object can require its own buffers and its own settings for attribute pointers.
- Before drawing each object, it would be necessary to call `gl.bindBuffer()` and `gl.vertexAttribPointer()` for each attribute.
- A typical 3D graphics program would use attributes for vertex coordinates, normal vectors, material properties, and texture coordinates. So, there would be a lot of function calls for each object.
- To help with this situation, WebGL 2.0 introduced **Vertex Array Objects (VAOs)**.
- A VAO is a section of memory, typically stored on the graphics card. It holds settings that are used by rendering functions such as `gl.drawArrays()`.
- This includes the enabled state of each attribute, references to the buffers used for the attribute data, and the values of all properties that are set by calling `gl.vertexAttribPointer()`.
- It also includes the settings and a reference to the buffer used by `gl.drawElements()`.

## Vertex Array Objects

- WebGL 2.0 has a default VAO, which it uses when no other VAO has been selected.
- To use an alternative VAO, we first have to create it, by calling `gl.createVertexArray()`:

```
vao = gl.createVertexArray();
```

- The return value `vao`, is an identifier for the VAO that has been created.
- In the new VAO, all properties have their default values. In particular, all vertex attributes are disabled and have no associated buffers.
- To actually use a VAO, we need to bind it:

```
gl.bindVertexArray(vao);
```

- Functions that affect or use attributes apply to the VAO that is currently bound. For example, the settings in a call to `gl.vertexAttribPointer()` are stored in the current VAO, and a call to `gl.drawArrays()` gets all the data that it needs to draw a primitive from the current VAO.
- A program can switch from one VAO to another at any time simply by calling `gl.bindVertexArray`.
- To go back to using the default VAO, a program can call `gl.bindVertexArray(0)`.



## Vertex Array Objects

- The idea is that a program that draws several objects can use a different VAO for each object.
- The VAO for an object must be bound when the settings for the object are configured.
- However, before drawing the object, the program simply needs to bind the VAO for that object.
- That single function call replaces a potentially a large number of function calls that would be needed to restore the appropriate settings for each attribute individually.
- The advantage is more than just a more nicely organized program because it is also much more efficient, since only one command needs to be sent to the GPU to configure all of the attributes!

## In-Class Demo

The following demo uses a different VAO for each of six different objects. The program uses many techniques that we have not yet covered, but you can look at the `drawModel()` function to see how it uses VAOs and VBOs.

[VAO test](#)



# Instanced Drawing

## Instanced Drawing

- It is common for a scene to contain multiple copies of the same primitive (that is, using the same vertex coordinates), but with different transformations, colors, or other properties for each copy.
- WebGL 2.0 makes it possible to draw all those copies with a single function call!
- This is called **instanced drawing** or **instancing**, and the individual copies of the primitive are called **instances**.
- The functions that use instanced drawing are `gl.drawArraysInstanced()` and `gl.drawElementsInstanced()`.
- **Instanced properties** (the properties that vary from one instance to another) are things that would likely be uniform variables when drawing each instance separately.
- Each instance gets just one value of the property that applies to all the vertices of the instance. Nevertheless, the properties are represented by attribute variables in the shader program, **not uniform variables**, and they are configured as attributes.

## Instanced Drawing

- To specify that an attribute is an instanced property, we just need to specify a "**divisor**" for that attribute.
- This is done by calling `gl.vertexAttribDivisor()`:  

```
g.vertexAttribDivisor(attribID, divisor);
```
- Here, `attribID` is the identifier for the attribute, as returned by `gl.getAttribLocation()`.
- The divisor is a non-negative integer. Passing zero as the divisor will turn off instancing for the attribute.
- If divisor is positive, then each value of the attribute will apply to that many instances. For example, if divisor is 3, then the first entry in the attribute value array applies to the first, second, and third instances; the second value in the array applies to the fourth, fifth, and sixth instances; and so on.
- In practice, the value of divisor is usually one, meaning that each instance has its own entry in the attribute value array.

## Instanced Drawing

- For an instanced property, in addition to setting the divisor, it is still necessary to enable the attribute, load data for it into a VBO, and configure it with `gl.vertexAttribPointer()`.
- Of course, it is also necessary to draw the primitive using `gl.drawArraysInstanced()` or `gl.drawElementsInstanced()`, and not with `gl.drawArrays()` or `gl.drawElements()`.
- This [demo](#) shows an example of instanced drawing. The program draws 30 colored disks, where a disk is approximated by a primitive of type `gl.TRIANGLE_FAN`.
- Three attributes are used: an attribute that holds the coordinates of the vertices, an instanced attribute that holds the colors for the disks, and an instanced attribute that holds a different translation for each disk.

## Instanced Drawing

- Another point of interest in the [previous example](#) is its use of vertex buffer objects. The disks can be animated.
- The disks move, but their colors don't change. Since the colors don't change, the color values for the disks are loaded into a VBO once, during program initialization.
- The usage parameter in `gl.bufferData()` is set to `gl.STATIC_DRAW` because the data will not be modified.
- However, since the disks are moving, the values for the translations of the disks have to change in each frame.
- Therefore, new data is loaded into the corresponding VBO for each frame, with usage `gl.STREAM_DRAW` because the data that is being loaded will only be used once.
- VAOs and instancing require WebGL 2.0, but the same functionality is available in many implementations of WebGL 1.0 as optional extensions.

## References

- Interactive Computer Graphics, a Top-Down Approach with WebGL, 7<sup>th</sup> Edition – Edward Angel and Dave Shreiner, Pearson.
- David J. Eck, Introduction to Computer Graphics, Version 1.3, August 2021. Notes and demos from this source are licensed under a [Creative Commons Attribution-Noncommercial-ShareAlike 4.0 License](#). Formatting and style changes have been made to adapt the content for presentations.





**Thank you.**  
**DOUGLAS**COLLEGE

**DOUGLAS**COLLEGE

# CMPT 2361

## Computer Graphics

Shahriar Khosravi, Ph.D.

Lecture 9

## Plan for this Lecture

- WebGL Context Options
- Introduction to GLSL
- The RGB Triangle in WebGL
- A Shape Stamper Example
- The Points Primitive
- WebGL Error Handling



DOUGLAS COLLEGE

# WebGL Context Options

## WebGL Context Options

- A WebGL graphics context is created by the function `canvas.getContext()`, where `canvas` is a reference to the `<canvas>` element where the graphics context will draw.
- This function takes an optional second parameter that can be used to set the value of certain options in the graphics context.
- The second parameter is only needed if we want to give a non-default value to at least one of the options.
- The parameter is a JavaScript object whose properties are the names of the options:

```
let options = {
 alpha: false,
 depth: false
};
gl = canvas.getContext("webgl", options); // (or "webgl2")
```

- All of the options are Boolean-valued.



## WebGL Context Options

- **alpha**: determines whether the drawing buffer has an alpha component.
  - This is the alpha component for the image canvas as a whole.
  - If there is an alpha component, then it is possible for pixels in the canvas to be transparent, letting the background (on the web page behind the canvas) show through.
  - The default value is **true**.
  - It is safe to set the value to **false**, if we we the canvas to be fully opaque.
  - Setting it to **false** does not stop us from doing alpha blending of the drawing color with the image color; the RGB color components can still be computed by blending.
  - However, setting the value to **false** is only necessary if our program outputs pixels with alpha component less than 1.0, and we don't want our image to blend with the background of the canvas.
  - Note that a graphics context with an alpha component might be handled more efficiently, because web pages use RGBA colors for their display.

## WebGL Context Options

- **depth**: determines whether a depth buffer is allocated.
  - The default value is `true`.
  - We only need a depth buffer if we enable the depth test.
  - The depth buffer is generally not needed for 2D graphics.
  - If our application doesn't need it, eliminating the depth buffer can save some memory in the GPU!
- **antialias**: is used to request that antialiasing be applied to the image.
  - A WebGL implementation might ignore the request, for example if antialiasing is not supported by the GPU.
  - The default value is `true`.
  - Antialiasing can improve the quality of an image, but it can also significantly increase the computation time.

## WebGL Context Options

- **preserveDrawingBuffer** : determines whether the contents of the drawing buffer are discarded after the image has been copied to the web page.
  - The default value is `false`.
  - The drawing buffer is internal to WebGL. Its contents only become visible on the screen when the web browser copies the image onto the web page.
  - The default value for **preserveDrawingBuffer** means that once that happens, WebGL can discard its own copy of the image, which allows the GPU to free up resources for other operations.
  - As long as our rendering functions completely redraw the image every time they called, the default is fine.
  - We should set the value to `true` only if we need to keep the image around, so that we can add to it incrementally over time.





**More on GLSL**

## More on GLSL ES 1.00

- This section discusses GLSL ES 1.00 only. Remember that it can be used with both WebGL 1.0 and WebGL 2.0.
- A vertex or fragment shader can contain global variable declarations, type definitions, and function definitions.
- One of the functions must be `main()`, which is the entry point for the shader; that is, it is the function that is called by the GPU to process the vertex or fragment.
- The `main()` function takes no parameters and does not return a value, so it takes the form:

```
void main() {
 .
 .
 .
}
```

## More on GLSL ES 1.00

- Control structures are limited.
- If-statements take the same form as in C, but some limitations are placed on the for loop syntax.
- While and do-while loops are **not** allowed.
- Data structures include arrays and structs with some limitations.
- GLSL's strength lies in its built-in data types and functions for working with vectors and matrices.
- We will for now focus on data types `float`, `vec2`, `vec3`, and `vec4`.
- These types represent, respectively, 1, 2, 3, or 4 floating point numbers.
- Variable declarations are similar to C:

```
attribute vec3 a_coords; // (only in vertex shader)
vec3 rgb;
float width, height;
uniform vec2 u_size;
varying vec4 v_color;
```

## More on GLSL ES 1.00

- Attribute, uniform, and varying variables are used for communication between JavaScript and the shader program and between the vertex shader and the fragment shader.
- In the examples on the previous slides, we used the prefixes "a\_", "u\_", and "v\_" in the names of the variables to help ourselves remember, but that is not required.
- It is common to construct a value for a vector from individual numbers or from shorter vectors. GLSL has a flexible notation for it. Using the variables declared in the above examples, we can write:

```
rgb = vec3(1.0, 0.7, 0.0); // construct a vec3 from constants
v_color = vec4(rgb, 1.0); // construct a vec4 from a vec3 and a constant
gl_Position = vec4(a_coords, 0.0, 1.0); // vec4 from a vec2 and 2 constants
```
- In the last assignment statement, **gl\_Position** is the special built-in variable that is used in the vertex shader to give the coordinates of the vertex.
- **gl\_Position** is of type **vec4**, requiring four numbers, because the coordinates are specified as homogeneous coordinates.
- The special variable **gl\_FragCoord** in the fragment shader is also a **vec4**, giving the coordinates of the pixel as homogeneous coordinates.
- **gl\_FragColor** is a **vec4**, giving the four RGBA color components for the pixel.

## More on GLSL ES 1.00

- A vertex shader needs, at a minimum, an attribute to give the coordinates of the vertex. For 2D drawing, it is natural for that attribute to be of type `vec2`.
- If we assume that the values for the attribute are already expressed in clip coordinates, then the complete source code for the vertex shader could be as simple as:

```
attribute vec2 coords;
void main() {
 gl_Position = vec4(coords, 0.0, 1.0);
}
```

- For a corresponding minimal fragment shader, we might simply draw everything in yellow.

```
precision mediump float;
void main() {
 gl_FragColor = vec4(1.0, 1.0, 0.0, 1.0);
}
```



DOUGLAS COLLEGE

# The RGB Triangle in WebGL

## The RGB Triangle in WebGL

- The WebGL Triangle Example on the Blackboard uses an **attribute** of type **vec2** to specify the coordinates of the vertices of the triangle.
- Coordinates range from -1 to 1 in the default WebGL coordinate system. For the triangle, the vertex coordinates are given in that range, so no coordinate transformation is needed.
- Since the color is different at each vertex of the triangle, the vertex color is also an **attribute**. We use an attribute of type **vec3** for the vertex colors, since no alpha component is needed in this program.
- The color of interior pixels in the triangle is interpolated from the colors at the vertices.
- The interpolation means that we need a **varying** variable to represent the color. A varying variable is assigned a value in the vertex shader, and its value is used in the fragment shader.
- We therefore need two color variables: an **attribute** and a **varying** variable. The varying variable carries the color from the vertex shader to the fragment shader. The shader just has to copy the value from the color attribute to the varying variable.



- The WebGL Triangle Example on the Blackboard uses an **attribute** of type **Vec2** to specify the coordinates of the vertices of the triangle.
  - Coordinates range from -1 to 1 for the triangle, the **gl\_Position** attribute is needed.
  - Since the color is constant for the triangle, we also need an **attribute**. We use **vec3** for the color, since we have no alpha component is not needed.
  - The color of interest is the color of the vertices.
  - The interpolation means that we need a **varying** variable to represent the color. A varying variable is assigned a value in the vertex shader, and its value is used in the fragment shader.
  - We therefore need two color variables: an **attribute** and a **varying** variable. The varying variable carries the color from the vertex shader to the fragment shader. The shader just has to copy the value from the color attribute to the varying variable.
- ```

attribute vec2 a_coords;
attribute vec3 a_color;
varying vec3 v_color;

void main() {
    gl_Position = vec4(a_coords, 0.0, 1.0);
    v_color = a_color;
}

```


The RGB Triangle in WebGL

- The fragment shader only has to copy the incoming color value from the **varying** variable into `gl_FragColor`, which specifies the outgoing color for the fragment:

```
precision mediump float;
varying vec3 v_color;

void main() {
    gl_FragColor = vec4(v_color, 1.0);
}
```

- In order to compile the shader program, the source code for the shaders has to be in JavaScript strings. For example, the fragment shader source code can be included in the JavaScript side as a global variable:

```
const fragmentShaderSource =
    "precision mediump float;\n" +
    "varying vec3 v_color;\n" +
    "void main() {\n" +
    "    gl_FragColor = vec4(v_color, 1.0);\n" +
    "}\n";
```

The RGB Triangle in WebGL

- Also on the JavaScript side, we need a global variable for the WebGL context. We also need to provide values for the attribute variables.
- We need global variables to represent the location of each attribute in the shader program, and to represent the VBOs that will hold the attribute values:

```
let gl; // The WebGL graphics context.  
let attributeCoords; // Location of the attribute named "a_coords".  
let bufferCoords; // A vertex buffer object to hold the values for a_coords.  
let attributeColor; // Location of the attribute named "a_color".  
let bufferColor; // A vertex buffer object to hold the values for a_color.
```

- The graphics context is created in the `init()` function. The other variables are initialized in a function `initGL()` that is called from `init()`. That function also creates the shader program, using the `createProgram()` function.

```
function initGL() {  
    let prog = createProgram( gl, vertexShaderSource, fragmentShaderSource );  
    gl.useProgram(prog);  
  
    attributeCoords = gl.getAttribLocation(prog, "a_coords");  
    bufferCoords = gl.createBuffer();  
  
    attributeColor = gl.getAttribLocation(prog, "a_color");  
    bufferColor = gl.createBuffer();  
}
```

The RGB Triangle in WebGL

- To set up the values for an attribute, we need six different JavaScript commands (and more if we count placing the attribute values into a typed array).
- The commands `getAttribLocation()` and `createBuffer()` will most likely be called just once for each attribute, so we will put them in an initialization routine.
- The other four commands are in `draw()`, the function that draws the image. In this program, `draw()` is called just once, so the division of the code into two functions is not really necessary, but in general, a draw function is meant to be called many times.
- It would be very inefficient to create a new VBO every time `draw()` is called.
- Before drawing the triangle, the `draw()` function fills the canvas with a black background.
- This is done using the WebGL functions `gl.clearColor()` and `gl.clear()`, which have exactly the same functionality as the OpenGL 1.1 functions `glClearColor()` and `glClear()`.

The RGB Triangle in WebGL

```
function draw() {
```

- To set up the WebGL context, we call `gl.getExtension('OES_element_index_uint')`.
- The context is created just once.
- The other program really runs in the browser.
- It would be better to use `gl.clearColor(0,0,0,1)` to clear the background to black.
- Before calling `gl.drawArrays`, we need to set up the vertex and color attributes.
- This is done exactly once in the `draw` function.

```
    gl.clearColor(0,0,0,1); // specify the color to be used for clearing
    gl.clear(gl.COLOR_BUFFER_BIT); // clear the canvas (to black)

    /* Set up values for the "a_coords" attribute */

    let coords = new Float32Array( [ -0.9,-0.8, 0.9,-0.8, 0,0.9 ] );

    gl.bindBuffer(gl.ARRAY_BUFFER, bufferCoords);
    gl.bufferData(gl.ARRAY_BUFFER, coords, gl.STREAM_DRAW);
    gl.vertexAttribPointer(attributeCoords, 2, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(attributeCoords);

    /* Set up values for the "a_color" attribute */

    let color = new Float32Array( [ 0,0,1, 0,1,0, 1,0,0 ] );

    gl.bindBuffer(gl.ARRAY_BUFFER, bufferColor);
    gl.bufferData(gl.ARRAY_BUFFER, color, gl.STREAM_DRAW);
    gl.vertexAttribPointer(attributeColor, 3, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(attributeColor);

    /* Draw the triangle. */

    gl.drawArrays(gl.TRIANGLES, 0, 3);
```

```
}
```

mands (and more if
most likely be called
e.
age. In this
functions is not
ry times.
lled.
black background.
ear()), which have

The RGB Triangle in WebGL

- In the `draw()` function, the variable `coords` contains values for the attribute named `"a_coords"` in the vertex shader.
- That attribute represents the x and y coordinates of the vertex. Since the attribute is of type `vec2`, two numbers are required for each vertex.
- The value for `coords` is created here with a `Float32Array` constructor that takes an ordinary JavaScript array as its parameter; the values from the JavaScript array are copied into the newly created typed array.
- Similarly, the variable `color` contains values for the `"a_color"` attribute in the vertex shader, with three numbers per vertex.

The Shape Stamper Example

The Shape Stamper Example

- This example is a simple interactive program where the user can place shapes in a canvas by clicking the canvas with the mouse.
- Properties of the shape are taken from a set of popup menus. The properties include the color and degree of transparency of the shape, as well as which of several possible shapes is drawn. The shape is centered at the point where the user clicks.
- In the RGB triangle example, color is an attribute, since a different color is assigned to each vertex of the triangle primitive.
- In the shape-stamper example, all vertices, and in fact all pixels, in a primitive have the same color. That means that color can be a **uniform** variable.
- The example also allows transparency, so colors need an alpha component as well as the RGB components.
- It is convenient to treat the alpha and RGB components as separate quantities, using two separate **uniform** variables in the shader program.
- The color and alpha uniforms are used in the fragment shader to assign the fragment's color.

The Shape Stamper Example

```
precision mediump float;
uniform vec3 u_color;
uniform float u_alpha;
void main() {
    gl_FragColor = vec4(u_color, u_alpha);
}
```

- To work with a **uniform** variable on the JavaScript side, we need to know its location in the shader program. The program gets the locations of the two uniform variables in the `intiGL()`:

```
uniformColor = gl.getUniformLocation(prog, "u_color");
uniformAlpha = gl.getUniformLocation(prog, "u_alpha");
```

- When a shape is drawn, the values from the popup menus determine the values of the uniforms:

```
let colorNumber = Number(document.getElementById("colorChoice").value);
let alpha = Number(document.getElementById("opacityChoice").value);
```

```
gl.uniform3fv( uniformColor, colorList[colorNumber] );
gl.uniform1f( uniformAlpha, alpha );
```


The Shape Stamper Example

- Values for uniform variables are set using the `gl.uniform*()` family of functions.
- In this case, `colorList[colorNumber]` is an array of three numbers holding the RGB color components for the color, so the function `gl.uniform3fv()` is used to set the value.
- The "3f" means that 3 floating point values are provided, and the "v" means that the three values are in an array.
- Note that three floating point values are required to match the type, `vec3`, of the uniform variable in the shader.
- The value of alpha is a single floating point number, so the corresponding uniform variable is set using `gl.uniform1f()`.
- In order for the alpha component of the color to have any effect, alpha blending must be enabled. This is done as part of initialization with the two commands

```
gl.enable( gl.BLEND );  
gl.blendFunc( gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA );
```

Enables use of the
alpha component.

How the alpha component is to be used.

The Shape Stamper Example

- When the program starts, the user sees a blank white canvas. When the user clicks the canvas, a shape is added.
- When the user clicks again, a second shape is added, but we want the first shape to remain. However, this is not the default behavior for WebGL.
- When the user clicks the canvas, an event-handler function for the **mousedown** event is called. The shape is drawn in that function.
- When the function returns, WebGL tells the web browser that the image has been modified, and the web browser copies the new image to the screen.
- Once that happens, the default behavior for WebGL is to discard the image. But this means that the second mouse click is drawing on a blank canvas, since the shape from the first mouse click has been erased.
- To fix this problem, the **preserveDrawingBuffer** option in the WebGL graphics context must be set to true.

The Shape Stamper Example

- When the program starts, the user sees a blank white canvas. When the user clicks the canvas, a shape is added.

```
let options = { // No need for alpha channel or depth buffer, but we
                // need to preserve the image in the drawing buffer.
                alpha: false,
                depth: false,
                preserveDrawingBuffer: true
            };
gl = canvas.getContext("webgl", options);
```

- Once that happens, the default behavior for WebGL is to discard the image. But this means that the second mouse click is drawing on a blank canvas, since the shape from the first mouse click has been erased.
- To fix this problem, the `preserveDrawingBuffer` option in the WebGL graphics context must be set to true.

The Shape Stamper Example

- Note that this program does not have a `draw()` function that redraws the entire image.
- All the drawing is done in the mouse-handling function, `doMouseDown()`.
- We could have been done this more efficiently. For example, the program could have used a data structure to store information about the shapes that have been drawn. Clicking the canvas would add an item to the list, and the entire image would then be redrawn, including the new shape.
- In this example, however, the only record of what's in the image is the image itself.
- WebGL uses a default coordinate system in which each of the coordinates ranges from `-1` to `1`.
- We normally would like to use a more convenient coordinate system, which means that we need to apply a coordinate transformation to transform the coordinates that we use into the default coordinate system.
- In this example, the x-coordinate ranges from `0` at the left to `canvas.width` at the right, and y ranges from `0` at the top to `canvas.height` at the bottom.

```
x2 = -1 + 2*( x1 / canvas.width );  
y2 = 1 - 2*( y1 / canvas.height );
```

The Shape Stamper Example

- In WebGL, the coordinate transformation is usually applied in the vertex shader.
- In this example, to implement the transformation, the vertex shader just needs to know the width and height of the canvas.
- The program provides the width and height to the vertex shader as **uniform** variables.
- The original pixel coordinates of the vertex are input to the vertex shader as an **attribute**.
- The shader applies the coordinate transformation to compute the value of **gl_Position**, which must be expressed in the default coordinate system:

```
attribute vec2 a_coords;    // pixel coordinates
uniform float u_width;      // width of canvas
uniform float u_height;     // height of canvas
void main() {
    float x = -1.0 + 2.0*(a_coords.x / u_width);
    float y = 1.0 - 2.0*(a_coords.y / u_height);
    gl_Position = vec4(x, y, 0.0, 1.0);
}
```

The Shape Stamper Example

- Transformations can be much more complicated than this, especially in 3D, but the general pattern holds:
 - Transformations are represented by uniform variables and are applied in the vertex shader.
- In general, transformations are implemented as matrices.
 - `uniform` variables can be matrices and the shader language GLSL has good support for matrix operations.
- In order to draw a shape, we need to store the coordinates for that shape in a `Float32Array`; then, we have to load the values from that array into the buffer associated with the `"a_coords"` attribute; and finally, we must call `gl.drawArrays()` to do the actual drawing.
- The coordinates for the shape can be computed based on what type of shape is being drawn and on the point where the user clicked. For example, the coordinate array for a circle is created by the following code, where `x` and `y` are the pixel coordinates for the point that was clicked.

The Shape Stamper Example

```
coords = new Float32Array(64);  
k = 0;  
for (let i = 0; i < 32; i++) {  
    let angle = i/32 * 2*Math.PI;  
    coords[k++] = x + 50*Math.cos(angle); // x-coord of vertex i  
    coords[k++] = y + 50*Math.sin(angle); // y-coord of vertex i  
}
```

- The circle is approximated as a 32-sided regular polygon, with a radius of 50 pixels. Two coordinates are required for each vertex, so the length of the array is 64.
- The code for the other shapes is similar. Once the array has been created, the shape is drawn using:

```
gl.bindBuffer(gl.ARRAY_BUFFER, bufferCoords);  
gl.bufferData(gl.ARRAY_BUFFER, coords, gl.STREAM_DRAW);  
gl.vertexAttribPointer(attributeCoords, 2, gl.FLOAT, false, 0, 0);  
gl.drawArrays(gl.TRIANGLE_FAN, 0, coords.length/2);
```

The number of vertices in the shape, since the array holds two numbers per vertex.



DOUGLAS COLLEGE

The Points Primitive

The Points Primitive

- This example demonstrates the `gl.POINTS` primitive.
- A `POINTS` primitive is a set of disconnected vertices. By default, each vertex is rendered as a single pixel. However, a program can specify a larger size.
- In OpenGL 1.1, this was done with the function `glPointSize()`. In WebGL, that function does not exist. Instead, the size is under the control of the vertex shader.
- When working on one of the vertices of a `POINTS` primitive, the vertex shader should assign a value to the special built-in variable `gl_PointSize`. The variable is of type `float`. It gives the size of the vertex, in pixels.
- The vertex is rendered as a square, centered at the vertex position, whose width and height are given by `gl_PointSize`.
- What this means is that the fragment shader will be called once for each pixel in that square.
- There is usually an implementation-dependent limit on the size of points. **The only size that is guaranteed to exist is one pixel**, but most implementations seem to support point sizes at least up to 64 pixels, and possibly much larger.

The Points Primitive

- When the fragment shader is called for a `POINTS` primitive, it is processing one pixel in the square of pixels surrounding the vertex.
- The special fragment shader variable `gl_PointCoord` tells the shader the location of the pixel within that square. The value of `gl_PointCoord` is an **input** to the shader.
- The type of `gl_PointCoord` is `vec2`, so it has two floating point components. The value of each component is in the range 0 to 1.
- The first component, `gl_PointCoord.x`, is 0 at the left edge of the square and 1 at the right.
- The second component, `gl_PointCoord.y`, is 0 at the top of the square and 1 at the bottom. So, for example, the value is (0,0) at the top-left corner, (1,0) at the top-right corner, and (0.5,0.5) at the center of the square.

The Points Primitive

- If the fragment shader uses `gl_PointCoord` in its computation, the color of the square can vary from pixel to pixel. As a simple example, setting

```
gl_FragColor = vec4( gl_PointCoord.x, 0.0, 0.0, 1.0 );
```

would render each vertex in the primitive as a square color gradient whose color varies horizontally from black on the left edge of the square to red on the right edge.

- In the example, we use `gl_PointCoord` to render the vertex as a disk instead of a square.
- The technique uses a new GLSL statement, `discard`, which is available only in the fragment shader. When the fragment shader executes the statement

```
discard;
```

the fragment shader terminates, and all further processing of the pixel is prevented. In particular, the color of the pixel in the image does not change.

- In the example, we use `discard` if the distance from `gl_PointCoord` to the center, (0.5,0.5), is greater than 0.5. This discards pixels that do not lie in the disk of radius 0.5.

The Points Primitive

- GLSL has a function for computing the distance between two vectors, so the test in the fragment shader is written like:

```
float distanceFromCenter = distance( gl_PointCoord, vec2(0.5,0.5) );  
if ( distanceFromCenter >= 0.5 ) {  
    discard; // don't draw this pixel!  
}
```
- In the program, the user can select whether the disks have random colors or are all colored red.
- Since each disk is a vertex of a single **POINTS** primitive, the fact that the disks can have different colors means that the color has to be given by an **attribute** variable.
- To implement random colors, a **Float32Array** is filled with random numbers, three for each vertex.
- The values are loaded into a VBO, and the values for the color attribute are taken from the VBO.

The Points Primitive

- But what happens when all the disks are red? Do we have to fill an array with multiple copies of "1, 0, 0" and use that data for the attribute?
- If we disable the `VertexAttribPointer` for the color attribute, then that attribute will have the same value for every vertex.
- The value is specified by the `gl.vertexAttrib*()` family of functions. So, in the example, the code for providing values for the color attribute is:

```
if ( randomColors ) {  
    // Use the attribute values from the color VBO,  
    //      which was filled during initialization.  
    gl.enableVertexAttribArray( attributeColor );  
}  
else {  
    // Turn off VertexAttribPointer,  
    //      and set a constant attribute color.  
    gl.disableVertexAttribArray( attributeColor );  
    gl.vertexAttrib3f( attributeColor, 1, 0, 0 );  
}
```

WebGL Error Handling

WebGL Error Handling

- In WebGL, an error such as an illegal parameter value will not in general crash the program or produce any automatic notification of the error.
- Instead, when WebGL detects such an error, it ignores the illegal function call, and it sets the value of an error code that gives some indication of the nature of the error.
- A program can check the current value of the error code by calling `gl.getError()`.
- This function returns an integer error code. The return value is `gl.NO_ERROR` if no error has occurred. Any other return value means that an error has occurred.
- Once an error code has been set, it stays set until `gl.getError()` is called, even if other, correct WebGL operations have been executed in the meantime.
- Calling `gl.getError()` retrieves the value of the error code and resets its value to `gl.NO_ERROR`.
- If we call `gl.getError()` twice in a row, the second call will always return `gl.NO_ERROR`.
- This means that when `gl.getError()` returns an error, the error might actually have been generated by an instruction that was executed some time ago.

WebGL Error Handling

- As an example, consider a call to `gl.drawArrays(primitive, first, count)`.
- If `primitive` is not one of the seven legal WebGL primitives, then WebGL will set the error code to `gl.INVALID_ENUM`.
- If `first` or `count` is negative, the error code is set to `gl.INVALID_VALUE`.
- If no shader program has been installed with `gl.useProgram()`, the error is `gl.INVALID_OPERATION`.
- If no data has been specified for an enabled vertex attribute, an error of type `gl.INVALID_STATE` occurs.
- These four error codes are the most common.
- It is impractical and inefficient to call `gl.getError()` after each WebGL function call. However, when something goes wrong, it can be used as a debugging aid:

```
console.log("Error code is " + gl.getError());
```


WebGL Error Handling

- Note that some browsers automatically output certain information about incorrect use of WebGL to their JavaScript console, which is part of the development tools built into many browsers.
- That console is also the destination for messages written using `console.log()`.
- It's always a good idea to check the console when running a WebGL program that is under development.

References

- Interactive Computer Graphics, a Top-Down Approach with WebGL, 7th Edition – Edward Angel and Dave Shreiner, Pearson.
- David J. Eck, Introduction to Computer Graphics, Version 1.3, August 2021. Notes and demos from this source are licensed under a [Creative Commons Attribution-Noncommercial-ShareAlike 4.0 License](#). Formatting and style changes have been made to adapt the content for presentations.



Thank you.
DOUGLASCOLLEGE

DOUGLASCOLLEGE

CMPT 2361

Computer Graphics

Shahriar Khosravi, Ph.D.

Lecture 10

Plan for this Lecture

- 2D Transforms in WebGL
- Transforms in JavaScript
- GLSL Basic Types
- GLSL Data Structures
- GLSL Qualifiers
- Function Definitions
- Control Structures
- Limits



DOUGLAS COLLEGE

2D Transforms in WebGL

2D Transforms in WebGL

- The basic 2D transforms are scaling, rotation, and translation. A sequence of such transformations can be combined into a single affine transform.
- A 2D affine transform maps a point (x_1, y_1) to the point (x_2, y_2) given by formulas of the form:

$$\begin{aligned}x_2 &= a*x_1 + c*y_1 + e \\ y_2 &= b*x_1 + d*y_1 + f\end{aligned}$$

- where $a, b, c, d, e,$ and f are constants. This transform can be represented as the 3-by-3 matrix:

$$\begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix}$$

- With this representation, a point (x, y) becomes the three-dimensional vector $(x, y, 1)$, and the transformation can be implemented as multiplication of the vector by the matrix.

2D Transforms in WebGL

- To apply a transformation to a primitive, each vertex of the primitive has to be multiplied by the transformation matrix.
- In GLSL, the natural place to do that is in the vertex shader. Technically, it would be possible to do the multiplication on the JavaScript side, but GLSL can do it more efficiently, since it can work on multiple vertices in parallel, and it is likely that the GPU has efficient hardware support for matrix math.
- It is a property of affine transformations that it suffices to apply them at the vertices of a primitive. Interpolation of the transformed vertex coordinates to the interior pixels of the primitive will give the correct result. This gives the same answer as interpolating the original vertex coordinates and then applying the transformation in the fragment shader.
- In GLSL, the type `mat3` represents 3-by-3 matrices, and `vec3` represents three-dimensional vectors.
- When applied to a `mat3` and a `vec3`, the multiplication operator `*` computes the product:

```
transformedCoords = transformMatrix * originalCoords;
```


2D Transforms in WebGL

- For 2D drawing, the original coordinates are likely to come into the vertex shader as an attribute of type `vec2`.
- We need to make the attribute value into a `vec3` by adding 1.0 as the z-coordinate.
- The transformation matrix is likely to be a uniform variable, to allow the JavaScript side to specify the transformation.
- This leads to the following minimal GLSL ES 1.00 vertex shader for working with 2D transforms.

```
attribute vec2 a_coords;  
uniform mat3 u_transform;  
void main() {  
    vec3 transformedCoords = u_transform * vec3(a_coords, 1.0);  
    gl_Position = vec4(transformedCoords.xy, 0.0, 1.0);  
}
```

2D Transforms in WebGL

- For 2D drawing, the original coordinates are likely to come into the vertex shader as an attribute of type `vec2`.
- We need to make the attribute value into a `vec3` by adding 1.0 as the z-coordinate.
- The transformation matrix is likely to be a uniform variable, to allow the JavaScript side to specify the transformation.
- This leads to the following minimal GLSL ES 1.00 vertex shader for working with 2D transforms.

```
attribute vec2 a_coords;  
uniform mat3 u_transform;  
void main() {  
    vec3 transformedCoords = u_transform * vec3(a_coords, 1.0);  
    gl_Position = vec4(transformedCoords.xy, 0.0, 1.0);  
}
```

For a GLSL ES 3.00 version, the "attribute" qualifier is replaced by "in", and a first line "#version 300 es" is added.

2D Transforms in WebGL

- The input coordinates are given as a `vec2`, (x,y), but we need a `vec3`, (x,y,1), to multiply by the matrix, so the first line of `main()` adds 1.0 as the z-coordinate.
- In the next line, the value for `gl_Position` must be a `vec4`. For a 2D point, the z-coordinate is 0.0, so we use only the x- and y-coordinates from `transformedCoords`.
- On the JavaScript side, the function `gl.uniformMatrix3fv()` is used to specify a value for a uniform of type `mat3`.
- To use it, the nine elements of the matrix should be stored in an array in column-major order.
- For loading an affine transformation matrix into a `mat3`, the command would be something like this:

```
gl.uniformMatrix3fv(u_transform_location, false, [ a, b, 0, c, d, 0, e, f, 1 ]);
```



DOUGLAS COLLEGE

Transforms in JavaScript

Transforms in JavaScript

- To work with transforms on the JavaScript side, we need a way to represent the transforms.
- We also need to keep track of a "current transform" that is the product all the individual modeling transformations that are in effect.
- The current transformation changes whenever a transformation such as rotation or translation is applied.
- We need a way to save a copy of the current transform before drawing a complex object and to restore it after drawing.
- Typically, a stack of transforms is used for that purpose. We are familiar with this pattern from both 2D (Canvas API) and 3D (OpenGL 1.1) graphics.
- The difference here is that the data structures and operations that we need are not built into the standard API, so we need some extra JavaScript code to implement them!

Transforms in JavaScript

- As an example, we can write a JavaScript class, `AffineTransform2D`, to represent affine transforms in 2D.
- The data for an object of type `AffineTransform2D` consists of the numbers `a`, `b`, `c`, `d`, `e`, and `f` in the transform matrix.
- There are methods in the class for multiplying the transform by scaling, rotation, and translation transforms.
- These methods modify the transform to which they are applied, by multiplying it on the right by the appropriate matrix.
- Here is a full description of the API, where `transform` is an object of type `AffineTransform2D`:
- `transform = new AffineTransform2D(a,b,c,d,e,f);` creates a `AffineTransform2D` with the matrix shown earlier.
- `transform = new AffineTransform2D();` creates an `AffineTransform2D` representing the identity transform.

Transforms in JavaScript

- `transform = new AffineTransform2D(original);` where `original` is an `AffineTransform2D`, creates a copy of `original`.
- `transform.rotate(r);` modifies `transform` by composing it with the rotation matrix for a rotation by `r` radians.
- `transform.translate(dx, dy);` modifies `transform` by composing it with the translation matrix for a translation by `(dx, dy)`.
- `transform.scale(sx, sy);` modifies `transform` by composing it with the scaling matrix for scaling by a factor of `sx` horizontally and `sy` vertically.
- `transform.scale(s);` does a uniform scaling, the same as `transform.scale(s, s)`.
- `array = transform.getMat3();` returns an array of nine numbers containing the matrix for `transform` in column-major order.

Transforms in JavaScript

- Note that an `AffineTransform2D` object does not represent an affine transformation as a matrix. Instead, it stores the coefficients `a`, `b`, `c`, `d`, `e`, and `f` as properties of the object.
- With this representation, the `scale` method in the `AffineTransform2D` class can be defined as follows:

```
scale(sx, sy = sx) { // Default value for sy is the value of sx.  
    this.a *= sx;  
    this.b *= sx;  
    this.c *= sy;  
    this.d *= sy;  
    return this;  
}
```

- This code multiplies the transform represented by the calling object by a scaling matrix, on the right.

Transforms in JavaScript

- Before a primitive is drawn, the current transform must be sent as a `mat3` into the vertex shader, where the `mat3` will be used to transform the vertices of the primitive.
- The method `transform.getMat3()` returns the transform as an array that can be passed to `gl.uniformMatrix3fv()`, which sends it to the shader program.
- To implement the stack of transformations, we can use an array of objects of type `AffineTransform2D`.
- In JavaScript, an array does not have a fixed length, and it comes with `push()` and `pop()` methods that make it possible to use the array as a stack.
- For convenience, we can define functions `pushTransform()` and `popTransform()` to manipulate the stack.
- In the simple hierarchy example on the Blackboard, the current transform is stored in a global variable named `transform`.

Transforms in JavaScript

- Before the transform is applied, the current transform is pushed onto the transform stack.
 - The transform stack is an array that stores the transforms that have been applied to the element.
 - To apply a transform, the current transform is pushed onto the transform stack.
 - In JavaScript, the transform stack is implemented as an array.
 - For each transform, the current transform is pushed onto the transform stack.
 - Here is the code for the transform stack:
- ```

let transform = new AffineTransform2D(); // Initially the identity.

const transformStack = []; // An array to serve as the transform stack.

/**
 * Push a copy of the current transform onto the transform stack.
 */
function pushTransform() {
 transformStack.push(new AffineTransform2D(transform));
}

/**
 * Remove the top item from the transform stack, and set it to be the current
 * transform. If the stack is empty, nothing is done (and there is no error).
 */
function popTransform() {
 if (transformStack.length > 0) {
 transform = transformStack.pop();
 }
}

```

## Transforms in JavaScript

- In this example, the function **square()** draws a square that has size 1 and is centered at (0,0) in its own object coordinate system.

```
function square() {
 gl.uniformMatrix3fv(u_transform_loc, false, transform.getMat3());
 gl.bindBuffer(gl.ARRAY_BUFFER, squareCoordsVBO);
 gl.vertexAttribPointer(a_coords_loc, 2, gl.FLOAT, false, 0, 0);
 gl.drawArrays(gl.LINE_LOOP, 0, 4);
}
```

- The coordinates for the square have been stored in a buffer, **squareCoordsVBO**, and **a\_coords\_loc** is the location of an attribute variable in the shader program. The variable **transform** holds the current modeling transform that must be applied to the square. It is sent to the shader program by calling:

```
gl.uniformMatrix3fv(u_transform_loc, false, transform.getMat3());
```

## Transforms in JavaScript

- The function `nestedSquares()` draws 16 squares.

```
function nestedSquares() {
 gl.uniform3f(u_color_loc, 0, 0, 1); // Set color to blue.
 square();
 for (let i = 1; i < 16; i++) {
 gl.uniform3f(u_color_loc, i/16, 0, 1 - i/16); // Red/Blue mixture.
 transform.scale(0.8);
 transform.rotate(framenumber / 200);
 square();
 }
}
```

- Between the squares, it modifies the modeling transform with:  

```
transform.scale(0.8);
transform.rotate(framenumber/200);
```
- The effect of these commands is cumulative, so that each square is a little smaller than the previous one, and is rotated a bit more than the previous one. The amount of rotation depends on the frame number in an animation.

## Transforms in JavaScript

- The function draws the nested squares centered at (0,0). In the main `draw()` routine, they are translated and made smaller:

```
pushTransform();
transform.translate(-0.5,0.5); // Move center of squares to (-0.5, 0.5).
transform.scale(0.85); // Reduce size from 1 to 0.85.
nestedSquares();
popTransform();
```

- The `pushTransform()` and `popTransform()` ensure that all of the changes made to the modeling transform while drawing the squares will have no effect on other objects that are drawn later.
- Transforms are, as always, applied to objects **in the opposite of the order in which they appear in the code**.



DOUGLAS COLLEGE

# GLSL Basic Types

## GLSL Basic Types

- Variables in GLSL must be declared before they are used. GLSL is a **strictly typed language**, and **every variable is given a type when it is declared**.
- GLSL has built-in types to represent scalars, vectors, and matrices.
- The scalar types are `float`, `int`, and `bool`. Version 3.00 adds an unsigned integer type, `uint`.
- A GPU might not support integers or Booleans on the hardware level, so it is possible that the `int` and `bool` types are actually represented as floating point values.
- The types `vec2`, `vec3`, and `vec4` represent vectors of two, three, and four floats.
- There are also types to represent vectors of ints (`ivec2`, `ivec3`, and `ivec4`) and bools (`bvec2`, `bvec3`, and `bvec4`).
- In version 3.00, there is also support for vectors of unsigned integers (`uvec2`, `uvec3`, and `uvec4`).
- GLSL has very flexible notation for referring to the components of a vector. One way to access them is with array notation. For example, `v[0]`, `v[1]`, `v[2]`, and `v[3]`.
- They can also be accessed using the dot notation as `v.x`, `v.y`, `v.z`, and `v.w`. The component names `x`, `y`, `z`, and `w` are appropriate for a vector that holds coordinates.
- Vectors can also be used to represent colors, and the components of `v` can alternatively be referred to as `v.r`, `v.g`, `v.b`, and `v.a`.
- Finally, they can be referred to as `v.s`, `v.t`, `v.p`, and `v.q`, names appropriate for texture coordinates.

## GLSL Basic Types

- GLSL allows us to use multiple component names after the dot, as in `v.rgb` or `v.zx` or even `v.yyy`. The **names can be in any order**, and **repetition is allowed**. This is called **swizzling**, and `v.zx` is an example of a **swizzler**.
- For example, if `v` is `vec4(1.0, 2.0, 3.0, 4.0)`, then `v.zx` is equivalent to `vec2(3.0, 1.0)`, and `v.yyy` is `vec3(2.0, 2.0, 2.0)`.
- Swizzlers can also be used on the left-hand side of an assignment. For example,  

```
vec4 coords = vec4(1.0, 2.0, 3.0, 4.0);
vec3 point = vec3(5.0, 6.0, 7.0);
coords.yzw = coords.wyz; // Now, coords is (1.0, 4.0, 2.0, 3.0)
point.xy = coords.xx; // Now, point is (1.0, 1.0, 7.0)
```
- A notation such as `vec2(1.0, 2.0)` is referred to as a "constructor," although it is not a constructor in the sense of C++, since GLSL is not object-oriented, and there is no `new` operator.



## GLSL Basic Types

- A constructor in GLSL consists of a type name followed by a list of expressions in parentheses, and it represents a value of the type specified by the type name.
- Any type name can be used, including the scalar types. The value is constructed from the values of the expressions in parentheses.
- An expression can contribute more than one value to the constructed value:

```
vec2 v = vec2(1.0, 2.0);
vec4 w = vec4(v, v); // w is (1.0, 2.0, 1.0, 2.0)
```

- Note that the expressions can be swizzlers:

```
vec3 v = vec3(1.0, 2.0, 3.0);
vec3 w = vec3(v.zx, 4.0); // w is (3.0, 1.0, 4.0)
```

## GLSL Basic Types

- Extra values from the last parameter will be dropped. This makes is possible to use a constructor to shorten a vector. However, it is not legal to have extra parameters that contribute no values at all to the result:

```
vec4 rgba = vec4(0.1, 0.2, 0.3, 0.4);
vec3 rgb = vec3(rgba); // takes 3 items from rgba; rgb is (0.1, 0.2, 0.3)
float r = float(rgba); // r is 0.1
vec2 v = vec2(rgb, rgba); // ERROR: No values from rgba are used.
```

- As a special case, when a vector is constructed from a single scalar value, all components of the vector will be set equal to that value:

```
vec4 black = vec4(1.0); // black is (1.0, 1.0, 1.0, 1.0)
```

## GLSL Basic Types

- When constructing one of the built-in types, type conversion will be applied if necessary.
- For purposes of conversion, the Boolean values true/false convert to the numeric values zero and one; in the other direction, zero converts to false and any other numeric value converts to true.
- Constructors are the only context in which GLSL does automatic type conversion. For example, we need to use a constructor to assign an int value to a float variable, and it is illegal to add an int to a float:

```
int k = 1;
float x = float(k); // OK; "x = k" would be a type mismatch error
x = x + 1.0; // OK
x = x + 1; // ERROR: Can't add values of different types.
```

## GLSL Basic Types

- The built-in matrix types are `mat2`, `mat3`, and `mat4`.
- The elements of a matrix can be accessed using array notation, such as `M[2][1]`. If a single index is used, as in `M[2]`, the result is a vector. For example, if `M` is of type `mat4`, then `M[2]` is a `vec4`.
- Arrays in GLSL, use column-major order. This means that `M[2]` is column number 2 in `M` rather than row number 2, and `M[2][1]` is the element in column 2 and row 1.
- A matrix can be constructed from the appropriate number of values, which can be provided as scalars, vectors or matrices. For example, a `mat3` can be constructed from nine `float` or from three `vec3` parameters:

```
mat3 m1 = mat3(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0);
vec3 v = vec3(1, 2, 3);
mat3 m2 = mat3(v, v, v);
```

- The matrix is filled in column-major order; that is, the first three numbers go into column 0, the next three into column 1, and the last three into column 2.

## GLSL Basic Types

- As a special case, if a matrix **M** is constructed from a single scalar value, then that value is put into all the diagonal elements of **M** (**M**[0][0], **M**[1][1], and so on).
- The non-diagonal elements are all set equal to **zero**. For example, `mat4(1.0)` constructs the four-by-four identity matrix.



# Data Structures

## Data Structures

- A GLSL program can define new types using the **struct** keyword.
- The syntax is the same as in C, with some limitations. A **struct** is made up of a sequence of named members, which can be of different types. The type of a member can be any of the built-in types, an array type, or a previously defined struct type. For example,

```
struct LightProperties {
 vec4 position;
 vec3 color;
 float intensity;
};
```

- This defines a type named **LightProperties**. The type can be used to declare variables:

```
LightProperties light;
```

- The members of the variable **light** are then referred to as **light.position**, **light.color**, and **light.intensity**.

## Data Structures

- Struct types have constructors, but their constructors do **not** support type conversion.
- The constructor must contain a list of values whose types exactly match the types of the corresponding members in the struct. For example,  

```
light = LightProperties(vec4(0.0, 0.0, 0.0, 1.0), vec3(1.0), 1.0);
```
- GLSL also supports arrays. Only one-dimensional arrays are allowed. The base type of an array can be any of the basic types or it can be a **struct** type.
- **The size of the array must be specified in the variable declaration as an integer constant.**

```
int A[10];
vec3 palette[8];
LightProperties lights[3];
```

- In version 1.00, **there are no array constructors, and it is not possible to initialize an array as part of its declaration.**



## Data Structures

- Version 3.00 does have array constructors, and it allows type names such as `int[10]`, representing an array of 10 integers:

```
int[4] B; // B is an array of 4 ints; GLSL ES 3.00 only!
```

```
B = int[4] (2, 3, 5, 7); // Array constructor; GLSL ES 3.00 only!
```

- Array indexing uses the usual syntax, such as `A[0]` or `palette[i+1]` or `lights[3].color`.
- In GLSL ES 1.00, there are some strong limitations on the expressions that can be used as array indices. With one exception, an expression that is used as the index for an array can contain only integer constants and for loop variables.
- For example, the expression `palette[i+1]` would only be legal inside a `for` of the form `for (int i = ...)`
- The single exception is that arbitrary index expressions can be used for arrays of uniforms in a vertex shader. Note that these restrictions do not apply in GLSL ES 3.00.
- Just as in C, there is no check for array index out of bounds errors. It is up to the programmer to make sure that array indices are valid.



DOUGLAS COLLEGE

# Qualifiers

## Qualifiers

- Variable declarations can be modified by various qualifiers, such as **attribute**, **uniform**, and **varying**. These are called **storage qualifiers**.
- The qualifiers **attribute** and **varying** do not exist in version 3.00; instead, an attribute is declared in the vertex shader using the storage qualifier **in**, and a varying variable is declared using **out** in the vertex shader and **in** in the fragment shader.
- The **uniform** qualifier is used in both versions. Only global variables, not local variables in function definition, can be **attribute**, **uniform**, or **varying** variables.
- The **attribute** qualifier can only be used in a GLSL ES 1.00 vertex shader, and it only applies to the built-in floating point types **float**, **vec2**, **vec3**, **vec4**, **mat2**, **mat3**, and **mat4**.
- In GLSL ES 3.00, the **in** qualifier on a vertex shader variable defines it to be an **attribute** variable, and it can be applied to integer and unsigned integer scalars and vectors, as well as to the floating point types.

## Qualifiers

- Also in GLSL ES 3.00, the **out** qualifier can be used on integer and floating point scalars and vectors in the fragment shader.
- In version 1.00, a fragment shader has the predefined variable **gl\_FragColor** of type **vec4** to specify the color of the pixel.
- In version 3.00, a fragment shader can have multiple outputs, and the outputs are not necessarily colors.
- Since the output type does not have to be **vec4**, it is not possible to have a predefined output variable.
- For now, we will only use one fragment shader output representing a color. So, a version 3.00 fragment shader will have one **out** variable of type **vec4**.
- When we discuss frame buffers, we will see why having multiple **out** variables in a fragment shader could be useful.

## Qualifiers

- Both the vertex shader and the fragment shader can use **uniform** variables. The same variable can occur in both shaders, as long as the types in the two shaders are the same.
- Uniform variables can be of any type, including array and structure types.
- JavaScript only has functions for setting **uniform** values that are scalar variables, vectors, or matrices.
- **There are no functions for setting the values of structs or arrays.** The solution to this problem requires treating every component of a **struct** or array as a separate uniform value. For example:

```
struct LightProperties {
 vec4 position;
 vec3 color;
 float intensity;
};
uniform LightProperties light[4];
```

## Qualifiers

- The variable `light` contains twelve basic values, which are of type `vec4`, `vec3`, or `float`.
- To work with the `light` uniform in JavaScript, we need twelve variables to represent the locations of the 12 components of the `uniform` variable.
- When using `gl.getUniformLocation()` to get the location of one of the 12 components, we need to give the full name of the component in the GLSL program. For example: `gl.getUniformLocation(prog, "light[2].color")`.
- It is natural to store the 12 locations in an array of JavaScript objects that parallels the structure of the array of structs on the GLSL side. Here is typical JavaScript code to create the structure and use it to initialize the `uniform` variables:

```
lightLocations = new Array(4);
for (i = 0; i < light.length; i++) {
 lightLocations[i] = {
 position: gl.getUniformLocation(prog, "light[" + i + "].position");
 color: gl.getUniformLocation(prog, "light[" + i + "].color");
 intensity: gl.getUniformLocation(prog, "light[" + i + "].intensity");
 };
}

for (i = 0; i < light.length; i++) {
 gl.uniform4f(lightLocations[i].position, 0, 0, 0, 1);
 gl.uniform3f(lightLocations[i].color, 1, 1, 1);
 gl.uniform1f(lightLocations[i].intensity, 0);
}
```

## Qualifiers

- For **uniform** shader variables that are matrices, the JavaScript function that is used to set the value of the **uniform** is `gl.uniformMatrix2fv()` for a `mat2`, `gl.uniformMatrix3fv()` for a `mat3`, or `gl.uniformMatrix4fv()` for a `mat4`.
- Although the matrix is two-dimensional, the values are stored in a one dimensional array. The values are loaded into the array in **column-major** order.
- For example, if `transform` is a **uniform mat3** in the shader, then JavaScript can set its value to be the identity matrix with:

```
transformLoc = gl.getUniformLocation(prog, "transform");
gl.uniformMatrix3fv(transformLoc, false, [1,0,0, 0,1,0, 0,0,1]);
```

- In Version 1.00, the second parameter must be `false`. In Version 3.00, the second parameter can be `true` to indicate that the entries of the matrix are provided in row-major rather than column-major order.
- Note that the 3 in `uniformMatrix3fv()` refers to the number of rows and columns in the matrix, not to the length of the array, which must be 9. It is also fine to use a typed array rather than a normal JavaScript array for the value of a uniform.

## Qualifiers

- A **varying** variable should be declared with the same name and type in both the vertex shader and fragment shader.
- In version 1.00, the storage qualifier for declaring varying variables is **varying**, and it can only be used for four the built-in floating point types (**float**, **vec2**, **vec3**, **vec4**, **mat2**, **mat3**, and **mat4**) and for arrays of those types.
- In version 3.00, a varying variable can also be an integer or unsigned integer scalar or vector. But there is a complication because it doesn't make sense to apply interpolation to integer values.
- So, a varying variable of integer type must be declared with the additional qualifier **flat**, which means it will **not** be interpolated. Instead, the value from the first vertex of a triangle or line segment will be used for every pixel. Floating point varying variables can also, optionally, be declared as flat. For example:

```
flat in ivec3 A; // GLSL ES 3.00 fragment shader only!
```

- Another possible storage qualifier is **const**, which means that the value of the variable cannot be changed after it has been initialized. The declaration of a **const** variable must include initialization.



## Qualifiers

- A variable declaration can also be modified by **precision qualifiers**.
  - `highp`, `mediump`, and `lowp`.
- A precision qualifier sets the minimum range of possible values for an integer variable or the minimum range of values and number of decimal places for a floating point variable.
- GLSL doesn't assign a definite meaning to the precision qualifiers, but mandates some minimum requirements.
  - For example, in version 1.00, `lowp` integers must be able to represent values in at least the range -28 to 28; `mediump` integers, in the range -210 to 210; and `highp` integers, in the range -216 to 216.
  - For version 3.00, `highp` variables always use 32 bits, and the requirements for `mediump` and `lowp` are higher.
- It is even possible that all values are 32-bit values and the precision qualifiers have no real effect. But GPUs in embedded systems can be more limited.

## Qualifiers

- A precision qualifier can be used on any variable declaration, including local variables and function parameters.
- If the variable also has a storage qualifier, **the storage qualifier comes first**. For example

```
lowp int n;
varying highp float v;
uniform mediump vec3 colors[3];
```

- A **varying** variable can have different precisions in the vertex and in the fragment shader.
- The default precision for integers and floats in the vertex shader is **highp**.
- Fragment shaders are **not** required to support **highp**.
- In the fragment shader, the default precision for integers is **mediump**, but **floats do not have a default precision**. This means that every floating point variable in the fragment shader has to be explicitly assigned a precision.
- Alternatively, it is possible to set a default precision for floats with the statement:

```
precision mediump float;
```

## Qualifiers

```
precision mediump float;
```

- If the GPU does support **highp** for fragment shaders, then the above statement restricts the precision.
- It is therefore helpful to test programmatically if support for **highp** exists in the fragment shader. We can use C-like preprocessor directives for this purpose at the beginning:

```
#ifdef GL_FRAGMENT_PRECISION_HIGH
 precision highp float;
#else
 precision mediump float;
#endif
```

- This sets the default precision to **highp** if it is available and to **mediump** if not.

## Qualifiers

- The next qualifier, **invariant**, and it has only a limited use.
- Invariance refers to the requirement that when the same expression is used to compute the value of the same variable (possibly in different shaders), then the value that is assigned to the variable should be exactly the same in both cases.
- This is **not** automatically the case.
- For example, the values can be different if a compiler uses different optimizations or evaluates the operands in a different order in the two expressions.
- The **invariant** qualifier on the variable will force the compiler to use exactly the same calculations for the two assignment statements.
- **The qualifier can only be used on declarations of varying variables.** It must be the first qualifier in the declaration:

```
invariant varying mediump vec3 color;
```

- It can also be used to make the predefined variables such as **gl\_Position** and **gl\_FragCoord** invariant, using a statement such as:

```
invariant gl_Position;
```



# GLSL Expressions

## Expressions

- Expressions in GLSL can use the arithmetic operators `+`, `-`, `*`, `/`, `++` and `--` for integer and floating point values.
- In version 3.00, the remainder operator, `%`, as well as left and right shift and bitwise logical operators, are also available for integer types.
- There is no automatic type conversion in expressions. If `x` is of type `float`, the expression `x+1` is **illegal**. We have to say `x+1.0` or `x+float(1)`.
- The arithmetic operators have been extended in various ways to work with vectors and matrices.
- If we use `*` to multiply a matrix and a vector, in either order, it multiplies them in the linear algebra sense, giving a vector as the result.
- The types of the operands must match in the obvious way; for example, a `vec3` can only be multiplied by a `mat3`, and the result is a `vec3`. When used with two matrices of the same size, `*` does matrix multiplication.

## Expressions

- If  $+$ ,  $-$ ,  $*$ , or  $/$  is used on a vector and a scalar of the same basic type, then the operation is performed on each element of the vector.
- For example, `vec2(3.0, 3.0) / 2.0` is the vector `vec2(1.5, 1.5)`, and `2*ivec3(1, 2, 3)` is the vector `ivec3(2, 4, 6)`.
- When one of these operators is applied to two vectors of the same type, the operation is applied to each pair of components, and the result is a vector. For example, the value of  

$$\text{vec3}(1.0, 2.0, 3.0) + \text{vec3}(4.2, -7.0, 1.7)$$
is the vector `vec3(5.2, -5.0, 4.7)`.
- The relational operators  $<$ ,  $>$ ,  $<=$ , and  $>=$  can only be applied to integer and floating point scalars, and the types of the two operands must match exactly.
- However, the equality operators `==` and `!=` have been extended to work on all of the built-in types.
- Two vectors are equal only if the corresponding pairs of components are all equal. The same is true for matrices. The equality operators cannot be used with arrays, but they do work for structs, as long as the structs don't contain any arrays; again, every pair of members in two structs must be equal for the structs to be considered equal.

## Expressions

- GLSL has logical operators `!`, `&&`, `||`, and `^^` (the last one being an exclusive or operation). The operands must be of type `bool`.
- Of course, the assignment operators `=`, `+=`, `-=`, `*=`, and `/=` have the usual meanings.
- GLSL also has a large number of predefined functions. All of the functions covered below require floating-point values as parameters, even if the function would also make sense for integer values.
- The function `dot(x, y)` computes the dot product  $x \cdot y$  of two vectors of the same length. The return value is a `float`.
- Function `cross(x, y)` computes the cross product  $x \times y$ , where the parameters and return value are of type `vec3`.
- Function `length(x)` is the length of the vector `x` and `distance(x, y)` gives the distance between two vectors.
- Function `normalize(x)` returns a unit vector that points in the same direction as `x`.



## Expressions

- The function `mix(x, y, t)` computes  $x*(1-t) + y*t$ . If `t` is a `float` in the range 0.0 to 1.0, then the return value is a linear mixture, or weighted average, of `x` and `y`. This function might be used, for example, to do alpha-blending of two colors.
- The function `clamp(x, low, high)` clamps `x` to the range `low` to `high`; the return value could be computed as `min(max(x, low), high)`.
- If `rgb` is a vector representing a color, we could ensure that all of the components of the vector lie in the range 0 to 1 with the command `rgb = clamp( rgb, 0.0, 1.0 );`
- If `s` and `t` are floats, with `s < t`, then `smoothstep(s, t, x)` returns 0.0 for `x` less than `s` and returns 1.0 for `x` greater than `t`. For values of `x` between `s` and `t`, the return value is smoothly interpolated from 0.0 to 1.0.

## Expressions

- Here is an example that might be used in a fragment shader for rendering a `gl.POINTS` primitive, with transparency enabled:

```
float dist = distance(gl_PointCoord, vec2(0.5));
float alpha = 1.0 - smoothstep(0.45, 0.5, dist);
if (alpha == 0.0) {
 discard; // discard fully transparent pixels
}
gl_FragColor = vec4(1.0, 0.0, 0.0, alpha);
```

- This would render the point as a red disk, with the color fading smoothly from opaque to transparent around the edge of the disk, as `dist` increases from 0.45 to 0.5.
- Note that for the functions `mix`, `clamp`, and `smoothstep`, the `x` and `y` parameters can be vectors as well as floats. In that case, they operate on each component of the vector individually.

## Expressions

- The usual mathematical functions are available in GLSL, including `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`, `log()`, `exp()`, `pow()`, `sqrt()`, `abs()`, `floor()`, `ceil()`, `min()`, and `max()`.
- For these functions, the parameters can be any of the types `float`, `vec2`, `vec3`, or `vec4`. The return value is of the same type, and the function is applied to each component separately.
- For example, the value of `sqrt(vec3(16.0, 9.0, 4.0))` is the vector `vec3(4.0, 3.0, 2.0)`.
- For `min()` and `max()`, there is also a second version of the function in which the first parameter is a vector and the second parameter is a `float`. For those versions, each component of the vector is compared to the `float`; for example, `max(vec3(1.0, 2.0, 3.0), 2.5)` is `vec3(2.5, 2.5, 3.0)`.
- The function `mod(x, y)` computes the modulus when `x` is divided by `y`. The return value is computed as `x - y*floor(x/y)`. As with `min` and `max`, `y` can be either a `vec` or a `float`. The `mod()` function can be used as a substitute for the `%` operator, which is not supported in GLSL ES 1.00.



DOUGLAS COLLEGE

# Function Definitions in GLSL

## Function Definitions

- A GLSL program can define new functions, with a syntax similar to C. Unlike C, function names can be overloaded.
- A function must be declared before it is used. As in C, it can be declared by giving either a full definition or a function prototype.
- Function parameters can be of any type. The return type for a function can be any type except for array types.
- A **struct** type can be a return type, as long as the structure does not include any arrays.
- When an array is used a formal parameter, the length of the array must be specified by an integer constant:

```
float arraySum10(float A[10]) {
 float sum = 0.0;
 for (int i = 0; i < 10; i++) {
 sum += A[i];
 }
 return sum;
}
```

## Function Definitions

- Function parameters can be modified by the qualifiers **in**, **out**, or **inout**. The default, if no qualifier is specified, is **in**.
- The qualifier indicates whether the parameter is used for input to the function, output from the function, or both.
- For input parameters, the value of the actual argument in the function call is copied into the formal parameter in the function definition, and there is no further interaction between the argument and parameters.
- For output parameters, the value of the actual parameter is copied to the formal argument when the function returns.
- For an **inout** parameter, the value is copied in both directions. This type of parameter passing is referred to as "call by value/return."
- Note that the actual parameter for an **out** or **inout** parameter must be something to which a value can be assigned, such as a variable or swizzler.
- GLSL has no pointers and does not support recursion.

## Function Definitions

- Function parameters can be modified by the qualifiers `in`, `out`, or `inout`. The default, if no qualifier is present, is `in`.
- The following code defines a function that calculates the cumulative sum of an array of floats. The function takes two arguments: a float array `A` of size 10, and a float array `B` of size 10. The function returns the value of `B[0]`, which is the first element of the output array.
- For an `in` parameter, the value of the argument is copied into the actual parameter when the function is called. This type of parameter passing is referred to as "call by value".
- For an `out` parameter, the value of the argument is copied to the actual parameter when the function returns. This type of parameter passing is referred to as "call by reference".
- For an `inout` parameter, the value is copied in both directions. This type of parameter passing is referred to as "call by value/reference".
- Note that the actual parameter for an `out` or `inout` parameter must be something to which a value can be assigned, such as a variable or swizzler.
- GLSL has no pointers and does not support recursions.

# Control Structures



## Control Structures

- The only control structures in GLSL ES 1.00 for WebGL are the **if** statement and a very restricted form of the **for** loop. There is no **while** or **do...while** loops, and there is no **switch** statement. However, all of these are supported in GLSL ES 3.00.
- If statements are supported with the full syntax from C, including **else** and **else if**. In version 3.00, the syntax for all control structures is very similar to C.
- In a **for** loop in a version 1.00 shader, the loop control variable must be declared in the loop, and it must be of type **int** or **float**.
- The initial value for the loop control variable must be a constant expression (all the operands must be literal constants or const variables).
- The code inside the loop is not allowed to change the value of the loop control variable. The test for ending the loop can only have the form **var op expression**, where **var** is the loop control variable, the **op** is one of the relational or equality operators, and the **expression** is a constant expression.
- The update expression must have one of the forms **var++**, **var--**, **var+=expression**, or **var-=expression**, where **var** is the loop control variable, and **expression** is a constant expression. Of course, this is the most typical form for **for** loops in other languages.

# Control Structures

- The only control structure that is not and a very restricted form of the `switch` statement is `for`.  
`for (int i = 0; i < 10; i++)`
- If statement: `if` statement. In version 3.00, the syntax is `if (expression) statement`.  
`for (float x = 1.0; x < 2.0; x += 0.1)`
- In a `for` loop in a version 1.00 standard, the loop control variable must be declared in the loop, and it must be of type `int` or `float`.  
`for (int k = 10; k != 0; k -= 1)`
- The **All loops can also include `break` and `continue` statements.** The operands must be literal constants or constant variables.
- The code inside the loop is not allowed to change the value of the loop control variable. The test for ending the loop can only have the form `var op expression`, where `var` is the loop control variable, the `op` is one of the relational or equality operators, and the `expression` is a constant expression.
- The update expression must have one of the forms `var++`, `var--`, `var+=expression`, or `var-=expression`, where `var` is the loop control variable, and `expression` is a constant expression. Of course, this is the most typical form for `for` loops in other languages.



DOUGLAS COLLEGE

# Limits

## Limits

- WebGL puts limits on certain resources that are used by WebGL and its GLSL programs, such as the number of **attribute** variables or the size of a texture image.
- The limits are due in many cases to hardware limits in the GPU, and they depend on the device on which the program is running, and on the implementation of WebGL on that device.
- The hardware limits can be lower on mobile devices such as tablets and phones, but modern tablets and phones have very impressive GPUs.
- Although the limits can vary, WebGL imposes a set of minimum requirements that all implementations must satisfy. We will cover the minimum requirements for WebGL 1.0. The minimum requirements for WebGL 2.0 are greater.
- For example, any WebGL implementation must allow at least 8 attributes in a vertex shader. The actual limit for a particular implementation might be more, but cannot be less.
- The actual limit is available in a GLSL program as the value of a predefined constant, **gl\_MaxVertexAttribs**. More conveniently, it is available on the JavaScript side as the value of the expression `gl.getParameter( gl.MAX_VERTEX_ATTRIBS );`

## Limits

- Attribute variables of type `float`, `vec2`, `vec3`, and `vec4` all count as one **attribute** against the limit.
- For a matrix-valued **attribute**, each column counts as a separate **attribute** as far as the limit goes.
- Similarly, there are limits on **varying** variables, and there are separate limits on **uniform** variables in the vertex and fragment shaders.
- The limits must satisfy:

```
gl_MaxVertexAttribs >= 8;
gl_MaxVertexUniformVectors >= 128;
gl_MaxFragmentUniformVectors >= 16;
gl_MaxVaryingVectors >= 8;
```

## Limits

- There are also limits in GLSL on the number of texture units, which means essentially the number of textures that can be used simultaneously. These limits must satisfy

```
gl_MaxTextureImageUnits >= 8; // limit for fragment shader
gl_MaxVertexTextureImageUnits >= 0; // limit for vertex shader
gl_MaxCombinedTextureImageUnits >= 8; // total limit for both shaders
```

- Note that `gl_MaxVertexTextureImageUnits` can be zero, which means that implementations are not required to allow texture units to be used in vertex shaders.
- There are also limits on other aspects, including viewport size, texture image size, line width for line primitives, and point size for the POINTS primitive. All of the limits can be queried from the JavaScript side using `gl.getParameter()`.

## References

- Interactive Computer Graphics, a Top-Down Approach with WebGL, 7<sup>th</sup> Edition – Edward Angel and Dave Shreiner, Pearson.
- David J. Eck, Introduction to Computer Graphics, Version 1.3, August 2021. Notes and demos from this source are licensed under a [Creative Commons Attribution-Noncommercial-ShareAlike 4.0 License](#). Formatting and style changes have been made to adapt the content for presentations.



**Thank you.**  
**DOUGLAS**COLLEGE

**DOUGLAS**COLLEGE



# CMPT 2361

## Computer Graphics

Shahriar Khosravi, Ph.D.

Lecture 11

## Plan for this Lecture

- 2D Textures in WebGL
  - Texture Units and Texture Objects
  - Working with Images
- Transformations in 3D
  - Using Backticks for Shader Programs
  - An Introduction to glmatrix
  - Transforming Coordinates
  - Transforming Normal Vectors

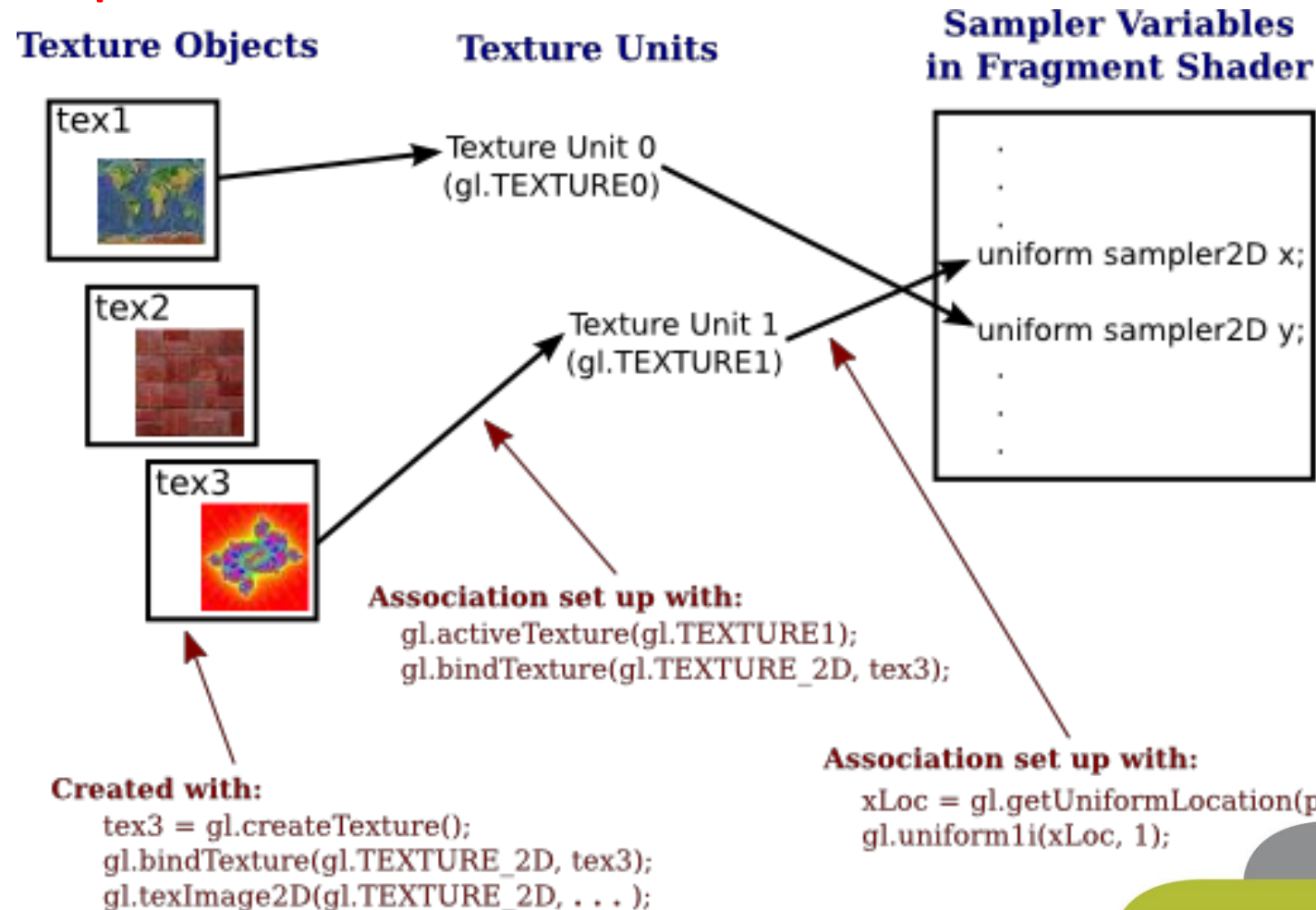


DOUGLAS COLLEGE

# Texture Units and Texture Objects

# Texture Units and Texture Objects

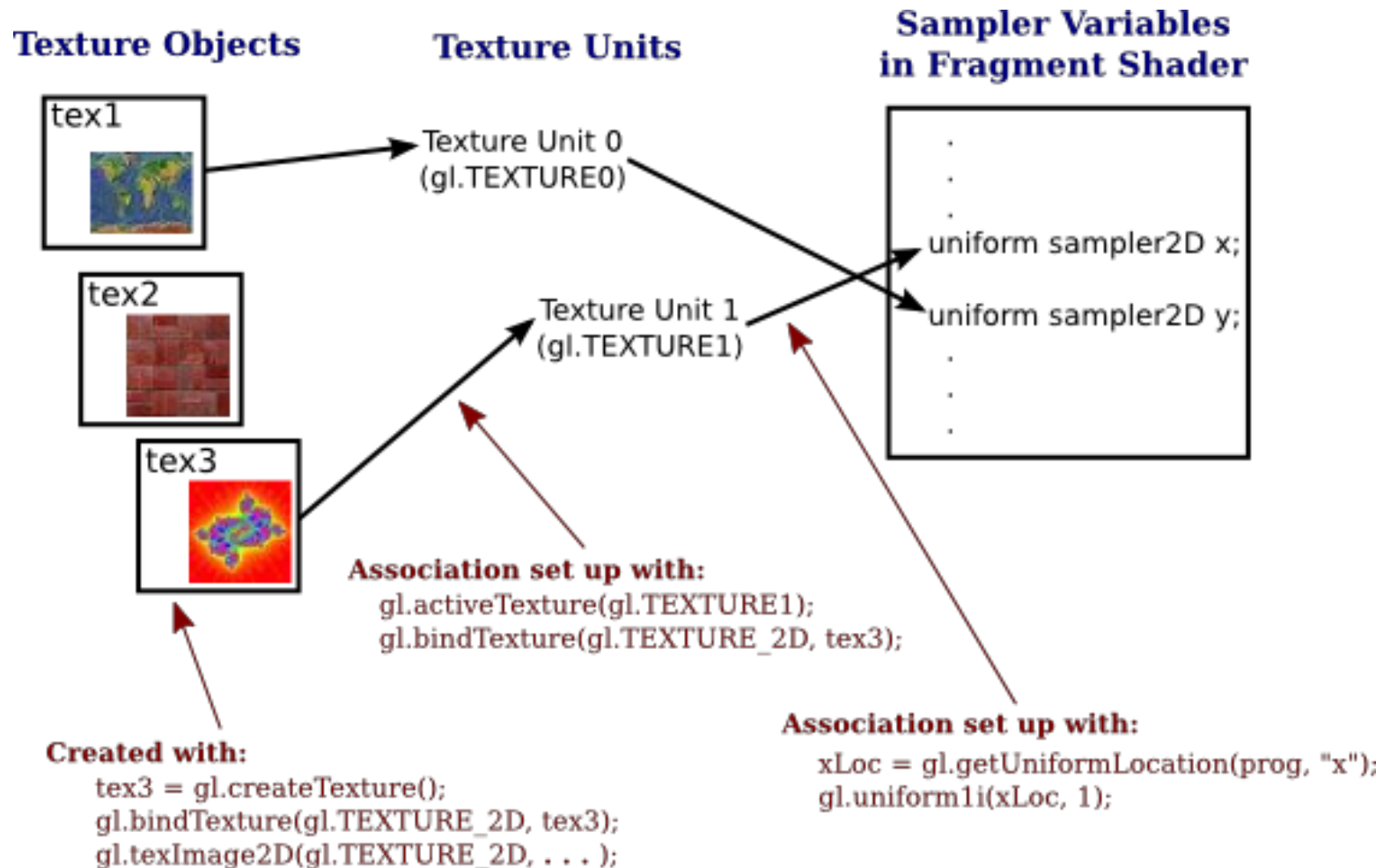
- Working with texture images in WebGL involves working with **texture objects**, **texture units**, and **sampler variables**:





# Texture Units and Texture Objects

- A **sampler variable** uses a **texture unit**, which uses a **texture object**, which holds a texture image.
- To apply a texture image to a primitive, we have to set up the entire chain.
- We also have to provide texture coordinates for the primitive, and use the sampler variable in the shader program to access the texture.



## Texture Units and Texture Objects

- A **texture unit**, also called a **texture mapping unit (TMU)** or a **texture processing unit (TPU)**, is a hardware component in a GPU that does **sampling**.
- **Sampling** is the process of computing a color from an image texture and texture coordinates.
- Mapping a texture image to a surface is a fairly complex operation, since it requires more than just returning the color of the texel that contains some given texture coordinates.
- It also requires applying the appropriate minification or magnification filter, possibly using mipmaps if available.
- Fast texture sampling is one of the key requirements for good GPU performance.
- **Texture units are not to be confused with texture objects.**
- A **texture object** is a data structure that contains the color data for an image texture, and possibly for a set of mipmaps for the texture, as well as the values of texture properties such as the minification and magnification filters and the texture repeat mode.
- A **texture unit** must access a texture object to do its work. The texture unit is the processor; the texture object holds the data that is processed.

## Texture Units and Texture Objects

- In GLSL, texture lookup is done using **sampler variables**.
- A **sampler variable** is a variable in a shader program.
- In GLSL ES 1.00, the only sampler types are **sampler2D** and **samplerCube**.
- A **sampler2D** is used to do lookup in a standard texture image; a **samplerCube** is used to do lookup in a [cubemap texture](#).
- The value of a sampler variable is a reference to a texture unit. The value tells which texture unit is invoked when the sampler variable is used to do texture lookup.
- Sampler variables must be declared as global **uniform** variables. It is not legal for a shader program to assign a value to a sampler variable. The value must come from the JavaScript side.
- On the JavaScript side, the available texture units are numbered 0, 1, 2, ..., where the maximum value is implementation dependent.

## Texture Units and Texture Objects

- The number of units can be determined as the value of the expression:

```
gl.getParameter(gl.MAX_COMBINED_TEXTURE_IMAGE_UNITS)
```

- As far as JavaScript is concerned, the value of a sampler variable is an integer.
- If we want a sampler variable to use texture unit number 2, then we set the value of the sampler variable to 2. This can be done using `gl.uniform1i()`.
- For example, suppose a shader program declares a sampler variable:

```
uniform sampler2D u_texture;
```

- To set its value from JavaScript, we need the location of the variable in the shader program:

```
u_texture_location = gl.getUniformLocation(prog, "u_texture");
```

- Then, we can tell the sampler variable to use texture unit number 2 by calling:

```
gl.uniform1i(u_texture_location, 2);
```

- Note that the integer value is not accessible in GLSL. The integer tells the sampler which texture unit to use, but there is no way for the shader program to find out the number of the unit that is being used.



## Texture Units and Texture Objects

- To use an image texture, we also need to create a texture object and load an image into the texture object.
- To set properties of the texture object, including mipmaps, and to associate the texture object with a texture unit, we use the JavaScript API.
- The command for creating a texture object is `gl.createTexture()`. This creates a single texture object and returns a reference to it. For example,

```
textureObj = gl.createTexture();
```

- This just allocates some memory for the object. To use it, we must first "bind" the texture object by calling `gl.bindTexture()`, like this:

```
gl.bindTexture(gl.TEXTURE_2D, textureObj);
```

- The parameter `gl.TEXTURE_2D`, is the texture target. This target is used for working with an ordinary texture image. There is a different target for cubemap textures.

## Texture Units and Texture Objects

- The function `gl.texImage2D()` is used to load an image into the currently bound texture object.
- We will revisit this command in a few slides, but it always applies to the currently bound texture object. The texture object is not mentioned in the command; instead, the texture object must be bound before the command is called.
- We also need to tell a texture unit to use the texture object. Before we can do that, we need to make the texture unit “active”, which is done by calling `gl.activeTexture()`.
- The input parameter is one of the constants `gl.TEXTURE0`, `gl.TEXTURE1`, `gl.TEXTURE2`, ..., which represent the available texture units.
- Initially, texture unit number 0 is active. To make texture unit number 2 active, for example, write:

```
gl.activeTexture(gl.TEXTURE2);
```

## Texture Units and Texture Objects

- We can then call

```
gl.bindTexture(gl.TEXTURE_2D, textureObj);
```

to bind a texture object, while texture unit 2 is active. The texture object `textureObj` is then bound to texture unit number 2 for `gl.TEXTURE_2D` operations.

- The binding just tells the texture unit which texture object to use. That is, when texture unit 2 does `TEXTURE_2D` lookups, it will do so using the image and the settings that are stored in `textureObj`.
- A texture object can be bound to several texture units at the same time. However, a given texture unit can have only one bound `TEXTURE_2D` at a time.

## Texture Units and Texture Objects

- Imagine we have several images to use on several different primitives. Between drawing primitives, we need to change the texture image that will be used. There are at least three different ways to manage the images in WebGL:
1. We could use a single texture object and a single texture unit.
    - The bound texture object, the active texture unit, and the value of the sampler variable can be set once and never changed.
    - To change to a new image, we would use `gl.texImage2D()` to load the image into the texture object.
    - This is very inefficient, except for when we use each image just once.
    - To eliminate this inefficiency, texture objects were introduced.

## Texture Units and Texture Objects

- Imagine we have several images to use on several different primitives. Between drawing primitives, we need to change the texture image that will be used. There are at least three different ways to manage the images in WebGL:
- 2. We could use a different texture object for each image, but use just a single texture unit.
  - The active texture and the value of the sampler variable will never have to be changed.
  - We would switch to a new texture image using `gl.bindTexture()` to bind the texture object that contains the desired image.

## Texture Units and Texture Objects

- Imagine we have several images to use on several different primitives. Between drawing primitives, we need to change the texture image that will be used. There are at least three different ways to manage the images in WebGL:
3. We could use a different texture unit for each image.
    - We would load each image into its own texture object and bind that object to one of the texture units.
    - We would switch to a new texture image by changing the value of the sampler variable.
- We are forced to use more than one texture unit only if we want to apply more than one texture image to the same primitive.
  - To do that, we will need several sampler variables in the shader program. They will have different values so that they refer to different texture units, and the color of a pixel will somehow depend on samples from both images (e.g., addition, or multiplication).

# Working with Images

## Working with Images

- An image can be loaded into a texture object using the function `gl.texImage2D()`

```
gl.texImage2D(target, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
```

`gl.TEXTURE_2D` for ordinary textures.

The mipmap level, which is 0 for the main image.

The format of the texture inside the texture object and in the original image. In WebGL 1.0, the two format parameters should have the same value.

We can use `gl.RGB` if we don't need the alpha component. Also, by using `gl.LUMINANCE` or `gl.LUMINANCE_ALPHA`, we can convert the image to grayscale.

**Luminance** is a weighted average of red, green, and blue that approximates the perceived brightness of a color.

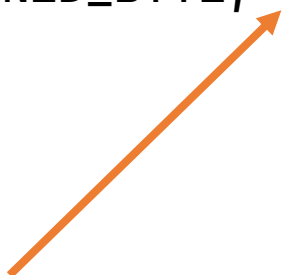
Always `gl.UNSIGNED_BYTE`, indicating that the colors in the image are stored using one byte for each color component.



## Working with Images

- An image can be loaded into a texture object using the function `gl.texImage2D()`  
`gl.texImage2D( target, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image );`

The image itself. Image can be a DOM image element that has been loaded asynchronously by JavaScript. The image can also be a `<canvas>` element. This means that we can draw on a canvas, using the HTML canvas 2D graphics API, and then use the canvas as the source for a texture image. We can even do that with an off-screen canvas that is not visible on the web page.



## Working with Images

- The image is loaded into the texture object that is currently bound to target in the currently active texture unit.
- There is no default texture object, so if no texture has been bound when `gl.texImage2D()` is called, an error occurs.
- The active texture unit is the one that has been selected using `gl.activeTexture()`, or is texture unit 0 if `gl.activeTexture()` has never been called.
- As discussed before, a texture object is bound to the active texture unit by `gl.bindTexture()`.
- Using images in WebGL is complicated by the fact that images are loaded asynchronously.
- We can specify a callback function that will be executed when the loading completes.
- When loading an image to use as a texture, the callback function should load the image into a texture object. Often, it will also call a rendering function to draw the scene, with the texture image.

## Working with Images

- In the triangle example on the Blackboard, there is a function that is used to load the texture image in that program.
- The texture object is created before the function is called.

```
function loadTexture(url, textureObject) {
 const img = new Image(); // A DOM image element to represent the image.
 img.onload = function() {
 // This function will be called after the image loads successfully.
 // We have to bind the texture object to the TEXTURE_2D target before
 // loading the image into the texture object.
 gl.bindTexture(gl.TEXTURE_2D, textureObject);
 try {
 gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, img);
 gl.generateMipmap(gl.TEXTURE_2D); // Create mipmaps; you must either
 // do this or change the minification filter.
 }
 catch (e) { // a security exception, because this page has been
 // loaded through a file:// URL.
 document.getElementById("headline").innerHTML =
 "Sorry, couldn't load texture.
" +
 "Some web browsers won't use images from a local disk";
 }
 draw(); // Draw the canvas, with or without the texture.
 };
 img.onerror = function() {
 // This function will be called if an error occurs while loading.
 document.getElementById("headline").innerHTML =
 "<p>Sorry, texture image could not be loaded.</p>";
 draw(); // Draw without the texture; triangle will be black.
 };
 img.src = url; // Start loading of the image.
 // This must be done after setting onload and onerror.
}
```

## Working with Images

- Note that image textures for WebGL 1.0 should be power-of-two textures.
- We can, however, use non-power-of-two textures, but we cannot use mipmaps with such textures, and the only texture repeat mode that is supported by such textures is `gl.CLAMP_TO_EDGE`. WebGL 2.0 does not have these restrictions.
- There are several parameters associated with a texture object, including the texture repeat modes and the minification and magnification filters.
- They can be set using the function `gl.texParameteri()`.
- The setting applies to the currently bound texture object. Most of the details are the same as in OpenGL 1.1.
- For example, the minification filter can be set to `LINEAR` using:

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
```

## Working with Images

- Recall that the default minification filter will not work without mipmaps.
- To get a working texture, we must change the minification filter or install a full set of mipmaps.
- Fortunately, WebGL has a function that will generate the mipmaps for us:

```
gl.generateMipmap(gl.TEXTURE_2D);
```

- The texture repeat modes determine what happens when texture coordinates lie outside the range 0.0 to 1.0. There is a separate repeat mode for each direction in the texture coordinate system.
- The possible values are `gl.REPEAT`, `gl.CLAMP_TO_EDGE`, and `gl.MIRRORED_REPEAT`. The default is `gl.REPEAT`.
- With `gl.MIRRORED_REPEAT`, the texture image is repeated to cover the entire plane, but every other copy of the image is reflected. This can eliminate visible seams between the copies. To set a texture to use mirrored repeat in both directions, use

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.MIRRORED_REPEAT);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.MIRRORED_REPEAT);
```

## Working with Images

- In WebGL, texture coordinates are input to the vertex shader as an **attribute** of type **vec2**.
- They are communicated to the fragment shader in a **varying** variable.
- The vertex shader will simply copy the value of the **attribute** into the **varying** variable.
- Another possibility is to apply an affine texture transformation to the coordinates in the vertex shader before passing them on to the fragment shader.
- In the fragment shader, the texture coordinates are used to sample a texture. The GLSL ES 1.00 function for sampling an ordinary texture is:

```
texture2D(samplerVariable, textureCoordinates);
```

where **samplerVariable** is the **uniform** variable of type **sampler2D** that represents the texture, and **textureCoordinates** is a **vec2** containing the texture coordinates.

- The return value is an RGBA color, represented as a value of type **vec4**.

## Working with Images

- In WebGL, texture coordinates are input to the vertex shader as an attribute of type `vec2`.
- They are declared in the vertex shader as follows:
 

```
precision mediump float;
uniform sampler2D u_texture;
varying vec2 v_texCoords;
```
- The vertex shader's main function is as follows:
 

```
void main() {
 vec4 color = texture2D(u_texture, v_texCoords);
 gl_FragColor = color;
}
```
- In the fragment shader, the texture coordinates are used to sample a texture. The GLSL ES 1.00 function for sampling an ordinary texture is:
 

```
vec4 texture2D(samplerVariable textureCoordinates);
```
- where `samplerVariable` is the uniform variable of type `sampler2D` that represents the texture, and `textureCoordinates` is a `vec2` containing the texture coordinates.
- The return value is an RGBA color, represented as a value of type `vec4`.

## Working with Images

- Textures are sometimes used on primitives of type `gl.POINTS`. In that case, we can get the texture coordinates for a pixel from the special fragment shader variable `gl_PointCoord`.
- A point is rendered as a square, and the coordinates in `gl_PointCoord` range from 0.0 to 1.0 over that square.
- Using `gl_PointCoord` here means that one copy of the texture will be pasted onto the point.
- If the `gl.POINTS` primitive has more than one vertex, we will see a copy of the texture at the location of each vertex.
- This is an easy way to put an image, or multiple copies of an image, into a scene. The technique is sometimes referred to as "**point sprites**."
- [This](#) demo draws a single textured primitive of type `gl.POINTS`, so we can see what it looks like. In the demo, only a circular cutout from each square point is drawn.



## Working with Images

- The pixel data for a texture image in WebGL is stored in memory starting with the row of pixels at the bottom of the image and working up from there.
- When WebGL creates the texture by reading the data from an image, it assumes that the image uses the same format. However, images in a web browser are stored in the opposite order, starting with the pixels in the top row of the image and working down. The result of this mismatch is that texture images will appear upside down.
- We can account for this by modifying our texture coordinates. However, we can also tell WebGL to invert the images for us as it "unpacks" them. To do that, call

```
gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, 1);
```

- We can do this as part of initialization. However, for `gl.POINTS` primitives, the coordinate system used by `gl_PointCoord` is already upside down, with the y-coordinate increasing from top to bottom.
- So, if we are loading an image for use on a `POINTS` primitive, we might want to set `gl.UNPACK_FLIP_Y_WEBGL` to its default value, 0.

# Shader Scripts

## Shader Scripts

- Thus far, we have created the source code strings by concatenating JavaScript string literals, one for each line of code.
- The example on the Blackboard uses this technique.
- There are at least two other techniques that are often used. One is to put the GLSL shader source code inside `<script>` elements. Here is an example for a vertex shader:

```
<script type="x-shader/x-vertex" id="vshader">
 attribute vec3 a_coords;
 uniform mat4 modelviewProjection;
 void main() {
 vec4 coords = vec4(a_coords,1.0);
 gl_Position = modelviewProjection * coords;
 }
</script>
```

## Shader Scripts

- This relies on the fact that a web browser will not recognize the type listed in the `<script>` element, so it will not try to execute the script.
- However, it does store the content of the `<script>` element in the DOM data structure that represents the web page.
- The content can be retrieved as a string using the standard DOM API. Here is a function that takes the id of the script element as its parameter and returns a string containing the text from inside the element:

```
function getTextContent(elementID) {
 let element = document.getElementById(elementID);
 let node = element.firstChild;
 let str = "";
 while (node) {
 if (node.nodeType == 3) // this is a text node
 str += node.textContent;
 node = node.nextSibling;
 }
 return str;
}
```

## Shader Scripts

- The other idea is to define the source code as a JavaScript template string.
- A template string is enclosed between single backquote characters and can span multiple lines.
- The "backquote" is also called a "backtick."
- Template strings were only introduced into JavaScript as part of ES6. They can include the values of JavaScript expressions, but we don't need that capability here. Here is how the above shader could be defined as a template string:

```
const vertexShaderSource = `
attribute vec3 a_coords;
uniform mat4 modelviewProjection;
void main() {
 vec4 coords = vec4(a_coords,1.0);
 gl_Position = modelviewProjection * coords;
}`;
```



DOUGLAS COLLEGE

# The glmatrix Library

## The glMatrix Library

- Transformations are essential to computer graphics. The WebGL API does not provide any functions for working with transformations.
- Last week, we used a simple JavaScript class to represent modeling transformations in 2D.
- For 3D graphics with WebGL, the JavaScript side will usually have to create both a modelview transform and a projection transform, and it will have to apply rotation, scaling, and translation to the modelview matrix, all without help from WebGL.
- Doing so is much easier if we have a JavaScript library to do the work.
- One commonly used library is **glMatrix**, a free JavaScript library for vector and matrix math written by Brandon Jones and Colin MacKenzie IV.
- It is available from <https://glmatrix.net>.

## The glMatrix Library

- The `glMatrix` API can be made available for use on a web page with a script element:  

```
<script src="gl-matrix-min.js"></script>
```
- This assumes that `gl-matrix-min.js` is in the same directory as the web page.
- The `glMatrix` library defines what it calls "classes" named `vec2`, `vec3`, and `vec4` for working with vectors of 2, 3, and 4 numbers.
- It defines `mat3` for working with 3-by-3 matrices and `mat4` for 4-by-4 matrices.
- The names should not be confused with the GLSL types of the same names; `glMatrix` is entirely on the JavaScript side. However, a `glMatrix mat4` can be passed to a shader program to specify the value of a GLSL `mat4`, and similarly for the other vector and matrix types.
- Each `glMatrix` "class" defines a set of functions for working with vectors and matrices.



## The glMatrix Library

- Although the documentation uses the term "class," `glMatrix` is not object-oriented. Its classes are just JavaScript objects. Vectors and matrices are represented in `glMatrix` as arrays, and the functions in classes like `vec4` and `mat4` simply operate on those arrays.
- There are no objects of type `vec4` or `mat4` as such, just arrays of length 4 or 16 respectively.
- The arrays can be either ordinary JavaScript arrays or typed arrays of type `Float32Array`.
- If we let `glMatrix` create the arrays for us, they will be `Float32Arrays`, but all `glMatrix` functions will work with either kind of array.
- For example, if the `glMatrix` documentation says that a parameter should be of type `vec3`, it is OK to pass either a `Float32Array` or a regular JavaScript array of three numbers as the value of that parameter.
- Note that it is also the case that either kind of array can be used in WebGL functions such as `gl.uniform3fv()` and `gl.uniformMatrix4fv()`.
- `glMatrix` is designed to work with those functions. For example, a `mat4` in `glMatrix` is an array of length 16 that holds the elements of a 4-by-4 array in column-major order, the same format that is used by `gl.uniformMatrix4fv()`.

## The glmMatrix Library

- Each `glmMatrix` class has a `create()` function which creates an array of the appropriate length and fills it with default values. For example,

```
transform = mat4.create();
```

sets `transform` to be a new `Float32Array` of length 16, initialized to represent the identity matrix.

- Similarly,

```
vector = vec3.create();
```

creates a `Float32Array` of length 3, filled with zeros.

- Each class also has a function `clone(x)` that creates a copy of its parameter `x`. For example:

```
saveTransform = mat4.clone(modelview);
```

## The glmMatrix Library

- Most other functions do not create new arrays. Instead, they modify the contents of their first parameter.
- For example, `mat4.multiply(A, B, C)` will modify `A` so that it holds the matrix product of `B` and `C`.
- Each parameter must be a `mat4` (i.e., an array of length 16) that already exists.
- It is OK for some of the arrays to be the same. For example, `mat4.multiply(A, A, B)` has the effect of multiplying `A` times `B` and modifying `A` so that it contains the answer.
- There are functions for multiplying a matrix by standard transformations such as scaling and rotation.
- For example if `A` and `B` are `mat4`s and `v` is a `vec3`, then `mat4.translate(A, B, v)` makes `A` equal to the product of `B` and the matrix that represents translation by the vector `v`.

## The glmMatrix Library

- In practice, we will use such operations mostly on a matrix that represents the modelview transformation.
- So, suppose that we have a `mat4` named `modelview` that holds the current `modelview` transform. To apply a translation by a vector `[dx,dy,dz]`, we can write:

```
mat4.translate(modelview, modelview, [dx,dy,dz]);
```

- This is equivalent to calling `glTranslatef(dx,dy,dz)` in OpenGL. So, if we draw some geometry after this statement, using `modelview` as the modelview transformation, then the geometry will first be translated by `[dx,dy,dz]` and then will be transformed by whatever was the previous value of `modelview`.
- Note the use of a vector to specify the translation in this command, rather than three separate parameters; this is typical of `glmMatrix`.
- To apply a scaling transformation with scale factors `sx`, `sy`, and `sz`, use:

```
mat4.scale(modelview, modelview, [sx,sy,sz]);
```

## The glMatrix Library

- For rotation, `glMatrix` has four functions, including three for the common cases of rotation about the x, y, or z axis.
- The fourth rotation function specifies the axis of rotation as the line from (0,0,0) to a point (dx,dy,dz).
- This is equivalent to `glRotatef(angle,dx,dy,dz)`. The angle of rotation in these functions is specified in radians rather than in degrees:

```
mat4.rotateX(modelview, modelview, radians);
mat4.rotateY(modelview, modelview, radians);
mat4.rotateZ(modelview, modelview, radians);
mat4.rotate(modelview, modelview, radians, [dx,dy,dz]);
```

- These functions allow us to do all the basic modeling and viewing transformations that we need for 3D graphics.

## The glMatrix Library

- To use hierarchical modelling, we also need to save and restore the transformation as we traverse the scene graph.
- For that, we need a stack. We can use a regular JavaScript array, which already has push and pop operations. So, we can create the stack as an empty array:

```
const matrixStack = [];
```

- We can then push a copy of the current modelview matrix onto the stack by writing:

```
matrixStack.push(mat4.clone(modelview));
```

- And we can remove a matrix from the stack and set it to be the current modelview matrix with `modelview = matrixStack.pop();`
- These operations are equivalent to `glPushMatrix()` and `glPopMatrix()` in OpenGL.

## The glmMatrix Library

- The starting point for the modelview transform is usually a viewing transform. In OpenGL, the function `gluLookAt()` is often used to set up the viewing transformation.
- The `glmMatrix` library has a `lookAt()` function to do the same thing:

```
mat4.lookAt(modelview, [eyex,eyey,eyez], [refx,refy,refz], [upx,upy,upz]);
```

- Note that this function uses three `vec3`s in place of the nine separate parameters in `gluLookAt()`, and it places the result in its first parameter instead of in a global variable.
- This function call is actually equivalent to the two OpenGL commands:

```
glLoadIdentity();
gluLookAt(eyex,eyey,eyez,refx,refy,refz,upx,upy,upz);
```

- So, we don't have to set `modelview` equal to the identity matrix before calling `mat4.lookAt()`, as we would usually do in OpenGL.
- However, we do have to create the `modelview` matrix at some point before using `mat4.lookAt()`, such as by calling:

```
let modelview = mat4.create();
```

## The glMatrix Library

- We can set an existing `mat4` to the identity matrix with the `mat4.identity()` function.

```
mat4.identity(modelview);
```

- We could use this as a starting point if we wanted to compose the view transformation out of basic scale, rotate, and translate transformations.
- Similarly, `glMatrix` has functions for setting up projection transformations. It has functions equivalent to `glOrtho()`, `glFrustum()`, and `gluPerspective()`, except that the field-of-view angle in `mat4.perspective()` is given in radians rather than degrees:

```
mat4.ortho(projection, left, right, bottom, top, near, far);
mat4.frustum(projection, left, right, bottom, top, near, far);
mat4.perspective(projection, fovyInRadians, aspect, near, far);
```

- As with the modelview transformation, we do not need to load projection with the identity before calling one of these functions, but we must create projection as a `mat4` (or an array of length 16).





DOUGLAS COLLEGE

# Transforming Coordinates

## Transforming Coordinates

- In WebGL, the transformation is usually done in the vertex shader.
- The coordinates for a primitive are specified in object coordinates. They are multiplied by the modelview transformation to convert them into eye coordinates and then by the projection matrix to convert them to the final clip coordinates that are actually used for drawing the primitive.
- Alternatively, the modelview and projection matrices can be multiplied together to get a matrix that represents the combined transformation; object coordinates can then be multiplied by that matrix to transform them directly into clip coordinates.
- In the shader program, coordinate transforms are usually represented as GLSL uniform variables of type `mat4`.
- The shader program can use either separate projection and modelview matrices or a combined matrix (or both).

## Transforming Coordinates

- Here is a minimal GLSL ES 1.00 vertex shader that uses a combined matrix:

```
attribute vec3 a_coords; // (x,y,z) object coordinates of vertex.
uniform mat4 modelviewProjection; // Combined transformation matrix.
void main() {
 vec4 coords = vec4(a_coords,1.0); // Add 1.0 for the w-coordinate.
 gl_Position = modelviewProjection * coords; // Transform the coordinates.
}
```

- For transformation matrices on the JavaScript side, we can use the `mat4` class from the `glMatrix` library to represent the projection, modelview, and combined transformation matrices:

```
const projection = mat4.create(); // projection matrix
const modelview = mat4.create(); // modelview matrix
const modelviewProjection = mat4.create(); // combined matrix
```

## Transforming Coordinates

- In the unlit cube example, only `modelviewProjection` corresponds to a shader variable. The location of that variable in the shader program is obtained during initialization using:

```
u_modelviewProjection = gl.getUniformLocation(prog, "modelviewProjection");
```

## Transforming Coordinates

- The transformation matrices are computed in the `draw()` function, using functions from the `glMatrix mat4` class.
- The value for `modelviewProjection` is sent to the shader program using `gl.uniformMatrix4fv()` before the primitives that make up the cube are drawn.

```

/* Set the value of projection to represent the projection transformation */
if (document.getElementById("persproj").checked)
 mat4.perspective(projection, Math.PI/5, 1, 4, 8);
else
 mat4.ortho(projection, -2, 2, -2, 2, 4, 8);

/* Set the value of modelview to represent the viewing transform. */
mat4.lookAt(modelview, [2,2,6], [0,0,0], [0,1,0]);

/* Apply the modeling tranformation to modelview. */
mat4.rotateX(modelview, modelview, rotateX);
mat4.rotateY(modelview, modelview, rotateY);
mat4.rotateZ(modelview, modelview, rotateZ);

/* Multiply the projection matrix times the modelview matrix to give the
 combined transformation matrix, and send that to the shader program. */
mat4.multiply(modelviewProjection, projection, modelview);
gl.uniformMatrix4fv(u_modelviewProjection, false, modelviewProjection);

```

## Transforming Coordinates

- If separate modelview and projection matrices are used in the shader program, then the modelview matrix can be applied to transform object coordinates to eye coordinates, and the projection can then be applied to the eye coordinates to compute `gl_Position`.

```
attribute vec3 a_coords; // (x,y,z) object coordinates of vertex.
uniform mat4 modelview; // Modelview transformation.
uniform mat4 projection; // Projection transformation
void main() {
 vec4 coords = vec4(a_coords,1.0); // Add 1.0 for w
 vec4 eyeCoords = modelview * coords; // Apply modelview transform.
 gl_Position = projection * eyeCoords; // Apply projection transform.
}
```



DOUGLAS COLLEGE

# Transforming Normal Vectors

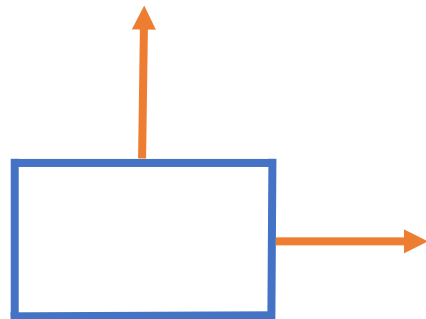
## Transforming Normal Vectors

- Normal vectors are essential for lighting calculations.
- When a surface is transformed in some way, it seems that the normal vectors to that surface will also change. However, that is not true if the transformation is a translation.
- A normal vector tells what direction a surface is facing. Translating the surface does not change the direction in which the surface is facing, so the normal vector remains the same.
- Our first guess might be that the normal vector should be transformed by just the rotation/scaling part of the transformation.
- This is like saying that the correct transformation is represented by the 3-by-3 matrix that is obtained by dropping the right column and the bottom row from the 4-by-4 coordinate transformation matrix.
  - The right column represents the translation part of the transformation, and the bottom row is only there because implementing translation in a matrix requires the use of homogeneous coordinates to represent vectors. Normal vectors, where translation is not an issue, do not use homogeneous coordinates.

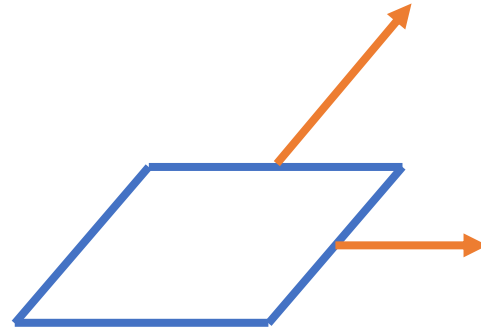


## Transforming Normal Vectors

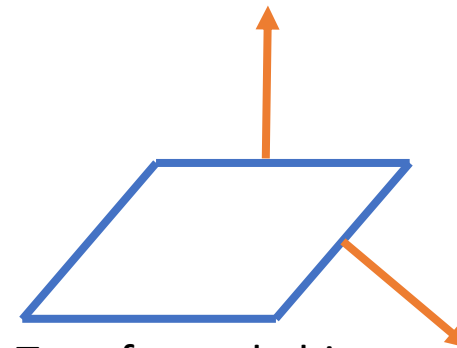
- However, this simplification won't be correct in all cases. As this illustration shows, if the normal vectors to an object are subjected to the same shear transformation as the object, the resulting vectors will not be perpendicular to the object:



Original quad  
with correct  
normal vectors.



Incorrect normal vectors as  
a result of applying the  
same transform to the  
normal vectors.



Transformed object  
with correct normal vectors.

## Transforming Normal Vectors

- Nevertheless, it is possible to get the correct transformation matrix for normal vectors from the coordinate transformation matrix.
- To do so, we must drop the fourth row and the fourth column and then take inverse transpose (i.e., the transpose of the inverse) of the resulting 3-by-3 matrix.
- If  $M$  is the current transformation matrix, then the rotated normal vector is given by:

$$N' = N \times M^{-1T}$$

- The `glmMatrix` library will compute this for us! The function that we need is `normalFromMat4()`, and it is defined in the `mat3` class:

```
mat3.normalFromMat4(normalMatrix, coordinateMatrix);
```

- Here, `coordinateMatrix` is the `mat4` that represents the transformation that is applied to coordinates, and `normalMatrix` is a `mat3` that already exists.
- This function computes the inverse transpose of the rotation/scale part of `coordinateMatrix` and places the answer in `normalMatrix`.
- Since we need normal vectors for lighting calculations, and lighting calculations are done in eye coordinates, the coordinate transformation that we are interested in is usually the modelview transform.

## Transforming Normal Vectors

- The normal matrix should be sent to the shader program, where it is needed to transform normal vectors for use in lighting calculations.
- Lighting requires unit normal vectors. The normal matrix does not in general preserve the length of a vector to which it is applied, so it will be necessary to normalize the transformed vector. GLSL has a built-in function for normalizing vectors.

```

attribute vec3 a_coords; // Untransformed object coordinates.
attribute vec3 normal; // Normal vector.
uniform mat4 projection; // Projection transformation matrix.
uniform mat4 modelview; // Modelview transformation matrix.
uniform mat3 normalMatrix; // Transform matrix for normal vectors.

.
. // Variables to define light and material properties.
.
void main() {
 vec4 coords = vec4(a_coords,1.0); // Add a 1.0 for the w-coordinate.
 vec4 eyeCoords = modelview * coords; // Transform to eye coordinates.
 gl_Position = projection * eyeCoords; // Transform to clip coordinates.
 vec3 transformedNormal = normalMatrix*normal; // Transform normal vector.
 vec3 unitNormal = normalize(transformedNormal); // Normalize.

 .
 . // Use eyeCoords, unitNormal, and light and material
 . // properties to compute a color for the vertex.
 .
}

```



**Thank you.**  
**DOUGLAS**COLLEGE

**DOUGLAS**COLLEGE

## References

- Interactive Computer Graphics, a Top-Down Approach with WebGL, 7<sup>th</sup> Edition – Edward Angel and Dave Shreiner, Pearson.
- David J. Eck, Introduction to Computer Graphics, Version 1.3, August 2021. Notes and demos from this source are licensed under a [Creative Commons Attribution-Noncommercial-ShareAlike 4.0 License](#). Formatting and style changes have been made to adapt the content for presentations.



# CMPT 2361

## Computer Graphics

Shahriar Khosravi, Ph.D.

Lecture 12

## Plan for this Lecture

- Lighting in WebGL
- Specular Reflection and Phong Shading
- More Complex Lighting
- Two-Sided Lighting
- Texture Transforms
- Procedural Textures



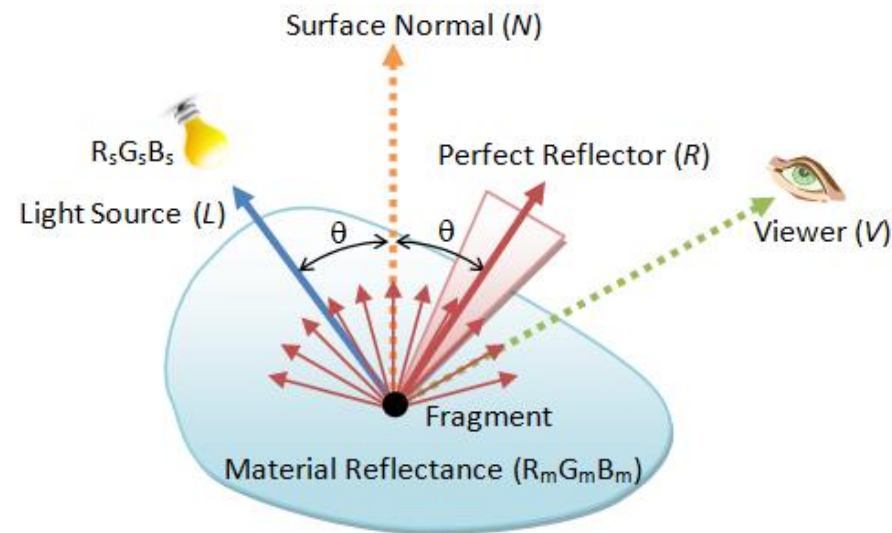
DOUGLAS COLLEGE

# Lighting in WebGL



## Lighting in WebGL

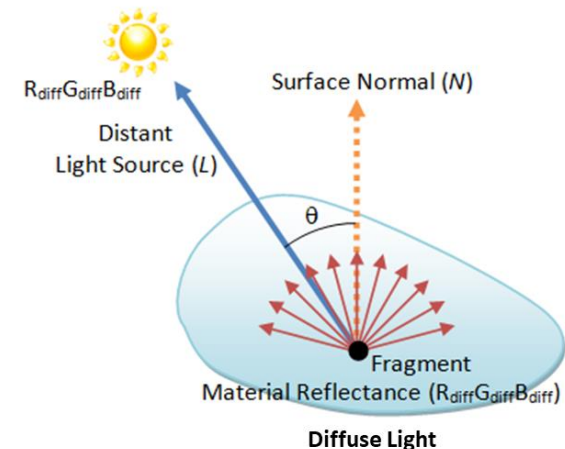
- The goal of the lighting equation is to compute a color for a point on a surface. The four vectors that are used in the computation are shown below:



- The vectors  $L$ ,  $N$ ,  $R$ , and  $V$  should be unit vectors. Recall that unit vectors have the property that the cosine of the angle between two unit vectors is given by the dot product of the two vectors.
- The lighting equation also involves ambient and emission color, which do not depend the direction vectors shown in the previous diagram.

## Lighting in WebGL

- Even very simple lighting, such as a "**viewpoint light**", can make 3D graphics look more realistic (i.e., light that shines from the direction of the viewer into the scene).
- In eye coordinates, a directional viewpoint light shines in the direction of the negative z-axis. The light direction vector ( $L$  in the previous diagram), which points towards the light source, is  $(0,0,1)$ .
- To keep things simple, consider diffuse reflection only. The color of the light reflected from a surface is the product of the diffuse material color of the surface, the color of the light, and the cosine of the angle at which the light hits the surface.
- The product is computed separately for the red, green, and blue components of the color. We are assuming that the light is white, so the light color is 1 in the formula.
- The material color will come from the JavaScript side as a **uniform** or **attribute** variable.



## Lighting in WebGL

- The cosine of the angle at which the light hits the surface is given by the dot product of the normal vector  $N$  with the light direction vector  $L$ .
- In eye coordinates,  $L$  is  $(0,0,1)$ . The dot product,  $N \cdot L$  or  $N \cdot (0,0,1)$ , is therefore simply  $N_z$ , the z-component of  $N$ . However, **this assumes that  $N$  is also given in eye coordinates**.
- The normal vector will ordinarily come from the JavaScript side and will be expressed in **object coordinates**. Before it is used in lighting calculations, it must be transformed to the eye coordinate system.
- As discussed in the previous lecture, to do that we need a normal transformation matrix that is derived from the modelview matrix.
- Since the normal vector must be of length one, the GLSL code for computing  $N$  would be something like: `vec3 N = normalize( normalMatrix * normal );`
- where `normal` is the original normal vector in object coordinates, `normalMatrix` is the normal transformation matrix, and `normalize()` is a built-in GLSL function that returns a vector of length one pointing in the same direction as its parameter.

## Lighting in WebGL

- There is one more complication: the dot product  $N \cdot L$  can be negative, which would mean that the normal vector points away from the light source (into the screen in this case).
- Ordinarily, that would mean that the light does not illuminate the surface. In the case of a viewpoint light, where we know that every visible surface is illuminated, it means that we are looking at the "back side" of the surface (or that incorrect normals were specified).
- Let's assume that we want to treat the two sides of the surface the same. The correct normal vector for the back side is the negative of the normal vector for the front side, and the correct dot product is  $(-N) \cdot L$ .
- We can handle both cases if we simply use  $\text{abs}(N \cdot L)$ . For  $L = (0,0,1)$ , that would be  $\text{abs}(N.z)$ .
- If `color` is a `vec3` giving the diffuse color of the surface, the visible color can be computed as:  
$$\text{vec3 visibleColor} = \text{abs}(N.z) * \text{color};$$

## Lighting in WebGL

- If `color` is instead a `vec4` giving an RGBA color, only the RGB components should be multiplied by the dot product:

```
vec4 visibleColor = vec4(abs(N.z)*color.rgb, color.a);
```

- The cube example on the Blackboard implements this simple lighting model. The lighting calculations are done in the vertex shader.
- Part of the scene is drawn without lighting, and the vertex shader has a uniform bool variable to specify whether lighting should be applied.
- It would be easy to add ambient light to this model, using a uniform variable to specify the ambient light level. Emission color is also easy.

# Lighting in WebGL

- If color is instead a vec4 giving an RGBA color, only the RGB components should be multiplied by the lighting.
- The calculation of the lighting is done in the fragment shader.
- Part of the lighting calculation is to specify the ambient color.
- It would be nice to specify the ambient color as a uniform bool variable.
- It would be nice to specify the ambient color as a uniform bool variable.
- It would be nice to specify the ambient color as a uniform bool variable.

```

attribute vec3 a_coords; // Object coordinates for the vertex.
uniform mat4 modelviewProjection; // Combined transformation matrix.
uniform bool lit; // Should lighting be applied?
uniform vec3 normal; // Normal vector (in object coordinates).
uniform mat3 normalMatrix; // Transformation matrix for normal vectors.
uniform vec4 color; // Basic (diffuse) color.
varying vec4 v_color; // Color to be sent to fragment shader.

void main() {
 vec4 coords = vec4(a_coords,1.0);
 gl_Position = modelviewProjection * coords;
 if (lit) {
 vec3 N = normalize(normalMatrix*normal); // Transformed unit normal.
 float dotProduct = abs(N.z);
 v_color = vec4(dotProduct*color.rgb, color.a);
 }
 else {
 v_color = color;
 }
}

```

## Lighting in WebGL

- The directional light used in the previous example is only “correct” for an orthographic projection, although it will generally give acceptable results for a perspective projection.
- The correct viewpoint light for a perspective projection is a point light at (0,0,0) (i.e., the position of the "eye" in eye coordinates).
- Remember that lighting calculations are done in eye coordinates. The vector L that points from the surface to the light can be computed as:

`vec3 L = normalize( lightPosition - eyeCoords.xyz );`

where `lightPosition` is a `vec3` that gives the position of the light in eye coordinates, and `eyeCoords` is a `vec4` giving the position of the surface point in eye coordinates.

- For a viewpoint light, the `lightPosition` is (0,0,0), and L can be computed simply as `normalize(-eyeCoords.xyz)`.

## Lighting in WebGL

- The eye coordinates for the surface point must be computed by applying the modelview matrix to the object coordinates for that point.
- This means that the shader program needs to know the modelview matrix; **it's not sufficient to know the combined modelview and projection matrix.**
- Our vertex shader can therefore be modified to use a point light at (0,0,0) :

```
attribute vec3 a_coords; // Object coordinates for the vertex.
uniform mat4 modelview; // Modelview transformation matrix
uniform mat4 projection; // Projection transformation matrix.
uniform bool lit; // Should lighting be applied?
uniform vec3 normal; // Normal vector (in object coordinates).
uniform mat3 normalMatrix; // Transformation matrix for normal vectors.
uniform vec4 color; // Basic (diffuse) color.
varying vec4 v_color; // Color to be sent to fragment shader.

void main() {
 vec4 coords = vec4(a_coords,1.0);
 vec4 eyeCoords = modelview * coords;
 gl_Position = projection * eyeCoords;
 if (lit) {
 vec3 L = normalize(- eyeCoords.xyz); // Points to light.
 vec3 N = normalize(normalMatrix*normal); // Transformed unit normal.
 float dotProduct = abs(dot(N,L));
 v_color = vec4(dotProduct*color.rgb, color.a);
 }
 else {
 v_color = color;
 }
}
```



# Specular Reflection and Phong Shading

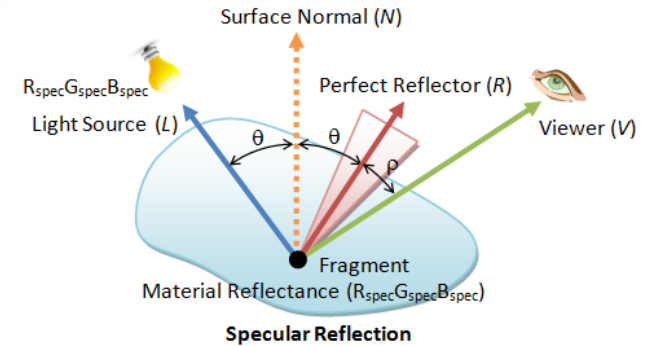
## Specular Reflection and Phong Shading

- To add specular lighting to our simple lighting model, we need to work with the vectors  $R$  and  $V$  in the lighting diagram.
- In the lighting equation that we are using, the amount of specular reflection depends on the dot product  $R \cdot V$ , which represents the cosine of the angle between  $R$  and  $V$ .
- The formula for the contribution of specular reflection to the visible color is:

$$(R \cdot V)^s * \text{specularMaterialColor} * \text{lightIntensity}$$

where  $s$  is the specular exponent (the "**shininess**").

- The unit vector  $R$  can be computed from  $L$  and  $N$  with some trigonometry as  $2 \cdot (N \cdot L) \cdot N - L$ .
- GLSL has a built-in function `reflect(I, N)` that computes the reflection of a vector  $I$  through a unit normal vector  $N$ ; however, the value of `reflect(L, N)` is  $-R$  rather than  $R$  because GLSL assumes a light direction vector that points from the light toward the surface, while our  $L$  vector does the reverse.



## Specular Reflection and Phong Shading

- The unit vector  $V$  points from the surface towards the position of the viewer.
- Remember that we are doing the calculations in eye coordinates. For an orthographic projection, the viewer is essentially at infinite distance, and  $V$  can be taken to be  $(0,0,1)$ .
- For a perspective projection, the viewer is at the point  $(0,0,0)$  in eye coordinates, and  $V$  is given by `normalize(-eyeCoords)` where `eyeCoords` contains the xyz coordinates of the surface point in the eye coordinate system.
- Putting all this together, and assuming that we already have  $N$  and  $L$ , the GLSL code for computing the color takes the form:

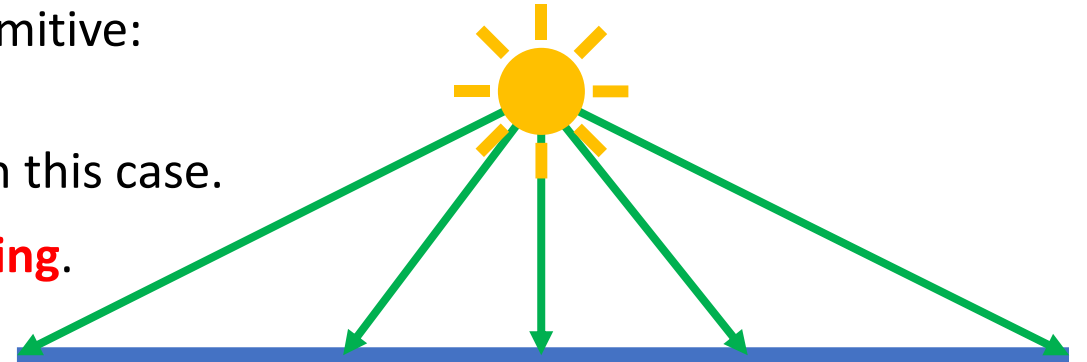
```
R = -reflect(L,N);
V = normalize(-eyeCoords.xyz); // (Assumes a perspective projection.)
vec3 color = dot(L,N) * diffuseMaterialColor.rgb * diffuseLightColor;
if (dot(R,V) > 0.0) {
 color = color + (pow(dot(R,V),specularExponent) *
 specularMaterialColor * specularLightColor);
}
```

## Specular Reflection and Phong Shading

- The specular lighting example implements lighting with diffuse and specular reflection.
- In this example, normal vectors are given as a vertex **attribute** rather than a **uniform** variable.
- To add some flexibility to the lighting, the light position is specified as a **uniform** variable rather than a constant.
- Following the OpenGL convention, **lightPosition** is a **vec4**. For a directional light, the w-coordinate is 0, and the eye coordinates of the light are **lightPosition.xyz**.
- If the w-coordinate is non-zero (usually 1), the light is a point light, and its eye coordinates are **lightPosition.xyz/lightPosition.w**.
- The example allows for different diffuse and specular material colors, but the light is always white, with diffuse intensity 0.8 and specular intensity 0.4.
- The fragment shader simply assigns the value of **v\_color** to **gl\_FragColor**.

## Specular Reflection and Phong Shading

- This approach imitates OpenGL 1.1 in that it does lighting calculations in the vertex shader. This is sometimes called **per-vertex lighting**.
- It is similar to Lambert shading in THREE.js, except that Lambert shading only uses diffuse reflection. But there are many cases where per-vertex lighting does not give good results.
- For example, it tends to produce bad specular highlights, unless the primitives are tiny.
- If a light source is close to a primitive, compared to the size of the primitive, then the angles that the light makes with the primitive at the vertices can have very little relationship to the angle of the light at an interior point of the primitive:
- Since lighting depends heavily on the angles, per-vertex lighting will not give a good result in this case.
- To get better results, we can do **per-pixel lighting**.
- This means we will move the lighting calculations from the vertex shader into the fragment shader.



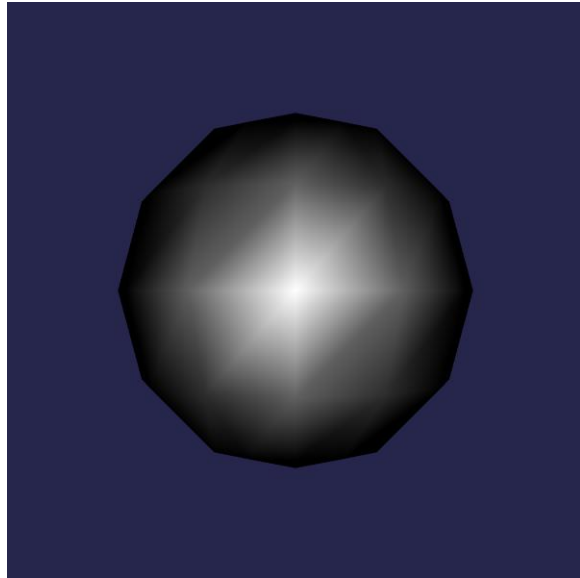
## Specular Reflection and Phong Shading

- To do per-pixel lighting, certain data that is available in the vertex shader must be passed to the fragment shader in **varying** variables.
  - For example, either object coordinates or eye coordinates for the surface point.
- The same might apply to properties such as diffuse color, if they are attributes rather than uniform variables.
- Uniform variables are directly accessible to the fragment shader. Light properties will generally be uniforms, and material properties might also be uniforms.
- Although normal vectors can sometimes be uniform variables, they are usually attributes.
- Per-pixel lighting generally uses interpolated normal vectors, passed to the fragment shader in a **varying** variable.
- An interpolated normal vector is in general only an approximation for the geometrically correct normal, but it is usually sufficient to produce good results.
- Interpolated normals are not necessarily unit vectors, even if the normals in the vertex shader are unit vectors. So, we must normalize the interpolated normal vectors in the fragment shader.
- The original normal vectors in the vertex shader should also be normalized, for the interpolation to work properly.

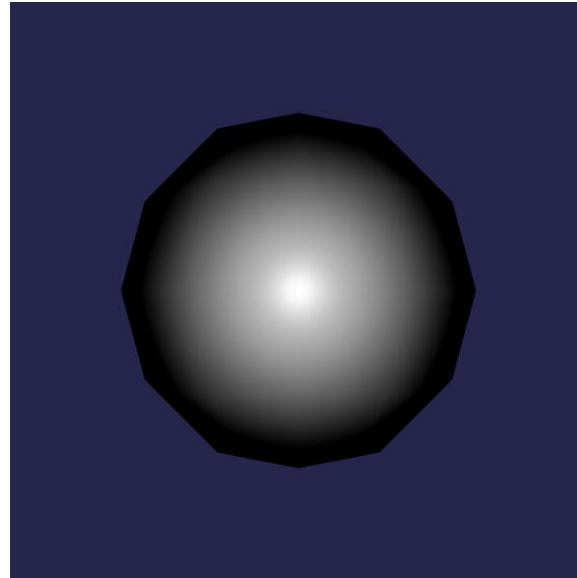
## Specular Reflection and Phong Shading

- The per-pixel-shading example uses per-pixel lighting.
- Aside from the fact that lighting calculations have been moved to the fragment shader, it is identical to the previous per-vertex shading example.

**Per-vertex Shading**



**Per-pixel Shading**





DOUGLAS COLLEGE

## More Complex Lighting



## Using Structures and Functions for Lighting

- It makes sense to define a **struct** to hold the properties of a light. The properties will usually include, at a minimum, the position and color of the light.
- Other properties can be added, depending on the application and the details of the lighting model that are used.
- For example, to make it possible to turn lights on and off, a **bool** variable might be added to say whether the light is enabled:

```
struct LightProperties {
 bool enabled;
 vec4 position;
 vec3 color;
};
```

- A light can then be represented as a variable of type **LightProperties**.
- It will likely be a **uniform** variable so that its value can be specified on the JavaScript side. Note that some of the parameters could be global variables in the shader program instead.

## Using Structures and Functions for Lighting

- Often, there will be multiple lights, represented by an array; for example, to allow for up to four lights: `uniform LightProperties lights[4];`
- Material properties can also be represented as a **struct**. Again, the details will vary from one application to another. For example, to allow for diffuse and specular color:

```
struct MaterialProperties {
 vec3 diffuseColor;
 vec3 specularColor;
 float specularExponent;
};
```

- The following function computes the contribution of one light to the color of a point on a surface:

## Using Structures and Functions for Lighting

```
vec3 lightingEquation(LightProperties light,
 MaterialProperties material,
 vec3 eyeCoords, // Eye coordinates for the point.
 vec3 N, // Normal vector to the surface.
 vec3 V // Direction to viewer.
) {
 vec3 L, R; // Light direction and reflected light direction.
 if (light.position.w == 0.0) { // directional light
 L = normalize(light.position.xyz);
 }
 else { // point light
 L = normalize(light.position.xyz/light.position.w - eyeCoords);
 }
 if (dot(L,N) <= 0.0) { // light does not illuminate the surface
 return vec3(0.0);
 }
 vec3 reflection = dot(L,N) * light.color * material.diffuseColor;
 R = -reflect(L,N);
 if (dot(R,V) > 0.0) { // ray is reflected toward the viewer
 float factor = pow(dot(R,V),material.specularExponent);
 reflection += factor * material.specularColor * light.color;
 }
 return reflection;
}
```

## Using Structures and Functions for Lighting

- Then, assuming that there are four lights, the full calculation of the lighting equation might look like this:

```
vec3 color = vec3(0.0); // Start with black (all color components zero).
for (int i = 0; i < 4; i++) { // Add in the contribution from light i.
 if (lights[i].enabled) { // Light can only contribute color if enabled.
 color += lightingEquation(lights[i], material,
 eyeCoords, normal, viewDirection);
 }
}
```

## Two-Sided Lighting

## Two-Sided Lighting

- In two-sided lighting, we need two materials, a front material for drawing front-facing polygons and a back material for drawing back-facing polygons.
- Furthermore, when drawing a back face, we have to reverse the direction of the normal vector, since normal vectors are assumed to point out of the front face.
- The fragment shader has a built-in Boolean variable named `gl_FrontFacing` whose value is set to `true` before the fragment shader is called, if the shader is working on the front face of a polygon.
- When doing per-pixel lighting, the fragment shader can check the value of this variable to decide whether to use the front material or the back material in the lighting equation.

## Two-Sided Lighting

```

uniform MaterialProperties frontMaterial;
uniform MaterialProperties backMaterial;
uniform LightProperties lights[3];
uniform mat3 normalMatrix;
varying vec3 v_normal;
varying vec3 v_eyeCoords;
A color for the fragment is computed using these variables and the lightingEquation function shown above:
vec3 normal = normalize(normalMatrix * v_normal);
vec3 viewDirection = normalize(-v_eyeCoords);
vec3 color = vec3(0.0);
for (int i = 0; i < 3; i++) {
 if (lights[i].enabled) {
 if (gl_FrontFacing) { // Computing color for a front face.
 color += lightingEquation(lights[i], frontMaterial, v_eyeCoords,
 normal, viewDirection);
 }
 else { // Computing color for a back face.
 color += lightingEquation(lights[i], backMaterial, v_eyeCoords,
 -normal, viewDirection);
 }
 }
}
gl_FragColor = vec4(color,1.0);

```

## Some Tips for One-Sided and Two-Sided Lighting

- Ordinarily, WebGL determines the front face of a triangle according to the rule that when the front face is viewed, vertices are listed in counterclockwise order around the triangle.
- The JavaScript command `gl.frontFace(gl.CW)` reverses the rule, so that vertices are listed in clockwise order when the front face is viewed.
- The command `gl.frontFace(gl.CCW)` restores the default rule.
- In some cases, we are sure that no back faces are visible. This will happen when the objects are closed surfaces seen from the outside, and all the triangles face towards the outside.
- So, it is wasted effort to draw back faces, since we know they will be hidden by front faces.
- The JavaScript command `gl.enable(gl.CULL_FACE)` tells WebGL to discard triangles without drawing them, based on whether they are front-facing or back-facing.
- The commands `gl.cullFace(gl.BACK)` and `gl.cullFace(gl.FRONT)` determine whether it is back-facing or front-facing triangles that are discarded when `CULL_FACE` is enabled; the default is to discard back-facing triangles.



## Some Tips for One-Sided and Two-Sided Lighting

- Displaying a set of grid lines on the surface or on its edges cause a problem with the depth test.
- OpenGL uses polygon offset to solve the problem. The same solution is available in WebGL. Polygon offset can be turned on with the commands:

```
gl.enable(gl.POLYGON_OFFSET_FILL);
gl.polygonOffset(1,1);
```

- Turn polygon offsetting off with:

```
gl.disable(gl.POLYGON_OFFSET_FILL);
```



DOUGLAS COLLEGE

# Texture Transforms

## Texture Transforms

- OpenGL maintains a texture transform matrix that can be manipulated to apply scaling, rotation, and translation to texture coordinates before they are used to sample a texture.
- It is easy to program the same operations in WebGL. We need to compute the texture transform matrix on the JavaScript side.
- The transform matrix is then sent to a **uniform** matrix variable in the shader program, where it can be applied to the texture coordinates.
- Note that as long as the texture transformation is affine, **it can be applied in the vertex shader, even though the texture is sampled in the fragment shader.**
- This means doing the transformation in the vertex shader and interpolating the transformed texture coordinates will give the same result as interpolating the original texture coordinates and applying the transformation to the interpolated coordinates in the fragment shader.

## Texture Transforms

- Since we are using `glMatrix` for coordinate transformation in 3D, it makes sense to use it for texture transforms as well.
- If we use 2D texture coordinates, we can implement scaling, rotation, and translation on the JavaScript side using the `mat3` class from `glMatrix`. The functions that we need are:
- `mat3.create()`: returns a new 3-by-3 matrix (represented as an array of length 9). The new matrix is the identity matrix.
- `mat3.identity(A)`: Sets `A` to be the identity matrix, where `A` is an already-existing `mat3`.
- `mat3.translate(A, B, [dx, dy])`: multiplies the matrix `B` by a matrix representing translation by `[dx, dy]`, and sets `A` to be the resulting matrix. `A` and `B` must already exist.
- `mat3.scale(A, B, [sx, sy])`: multiplies `B` by a matrix representing scaling by `[sx, sy]`, and sets `A` to be the resulting matrix.
- `mat3.rotate(A, B, angle)`: multiplies `B` by a matrix representing rotation by `angle` **radians** about the origin, and sets `A` to be the resulting matrix.

## Texture Transforms

- For implementing texture transformations, the parameters A and B in these functions will be the texture transform matrix.
- For example, to apply a scaling by a factor of 2 to the texture coordinates, we might use the code:  

```
var textureTransform = mat3.create();
mat3.scale(textureTransform, textureTransform, [2,2]);
gl.uniformMatrix3fv(u_textureTransform, false, textureTransform);
```
- The last line assumes that `u_textureTransform` is the location of a **uniform** variable of type `mat3` in the shader program.
- The moving texture example uses texture transformations to animate textures. In the program, texture coordinates are input into the vertex shader as an **attribute** named `a_texCoords` of type `vec2`, and the texture transformation is a **uniform** variable named `textureTransform` of type `mat3`.
- The transformed texture coordinates are computed in the vertex shader with the GLSL commands:

```
vec3 texcoords = textureTransform * vec3(a_texCoords,1.0);
v_texCoords = texcoords.xy;
```



DOUGLAS COLLEGE

# Procedural Textures

## Procedural Textures

- Up until now, all of our textures have been image textures. With an image texture, a color is computed by sampling the image, based on a pair of texture coordinates.
- The image essentially defines a function that takes texture coordinates as input and returns a color as output. However, there are other ways to define such functions besides looking up values in an image.
- A procedural texture is defined by a function whose value is computed rather than looked up. That is, the texture coordinates are used as input to a code segment whose output is the corresponding color value for the texture.
- In WebGL, procedural textures can be defined in the fragment shader. The idea is simple: Take a **vec2** representing a set of texture coordinates. Then, instead of using a **sampler2D** to look up a color, use the **vec2** as input to some mathematical computation that computes a **vec4** representing a color.
- In theory any computation could be used, as long as the components of the **vec4** are in the range 0.0 to 1.0.

## Procedural Textures

- Here is an example of a procedural texture:

```
void main()
{
 vec2 texCoords = v_texCoords;
 vec4 color;
 float a = floor(v_texCoords.x * 4.0);
 float b = floor(v_texCoords.y * 4.0);
 if (mod(a+b, 2.0) > 0.5)
 { // a+b is odd
 color = vec4(1.0, 0.0, 0.0, 1.0); // red
 }
 else
 { // a+b is even
 color = vec4(0.0, 0.0, 1.0, 1.0); // blue
 }
 gl_FragColor = color; // Just use the color from the texture as the pixel color.
}
```



## Procedural Textures

- The `floor()` function computes the largest integer less than or equal to its parameter.
- The value of `mod(a+b, 2.0)` is either 0.0 or 1.0, so the if-statement tests whether `a+b` is even or odd.
- The idea here is that when either `a` or `b` increases or decreases by 1, `a+b` will change from even to odd or from odd to even; that ensures that neighboring squares in the pattern will be differently colored.



**Thank you.**  
**DOUGLAS**COLLEGE

**DOUGLAS**COLLEGE

## References

- Interactive Computer Graphics, a Top-Down Approach with WebGL, 7<sup>th</sup> Edition – Edward Angel and Dave Shreiner, Pearson.
- David J. Eck, Introduction to Computer Graphics, Version 1.3, August 2021. Notes and demos from this source are licensed under a [Creative Commons Attribution-Noncommercial-ShareAlike 4.0 License](#). Formatting and style changes have been made to adapt the content for presentations.

# CMPT 2361

## Computer Graphics

Shahriar Khosravi, Ph.D.

Lecture 13

## Plan for this Lecture

- Bump Maps
- Framebuffers



DOUGLAS COLLEGE

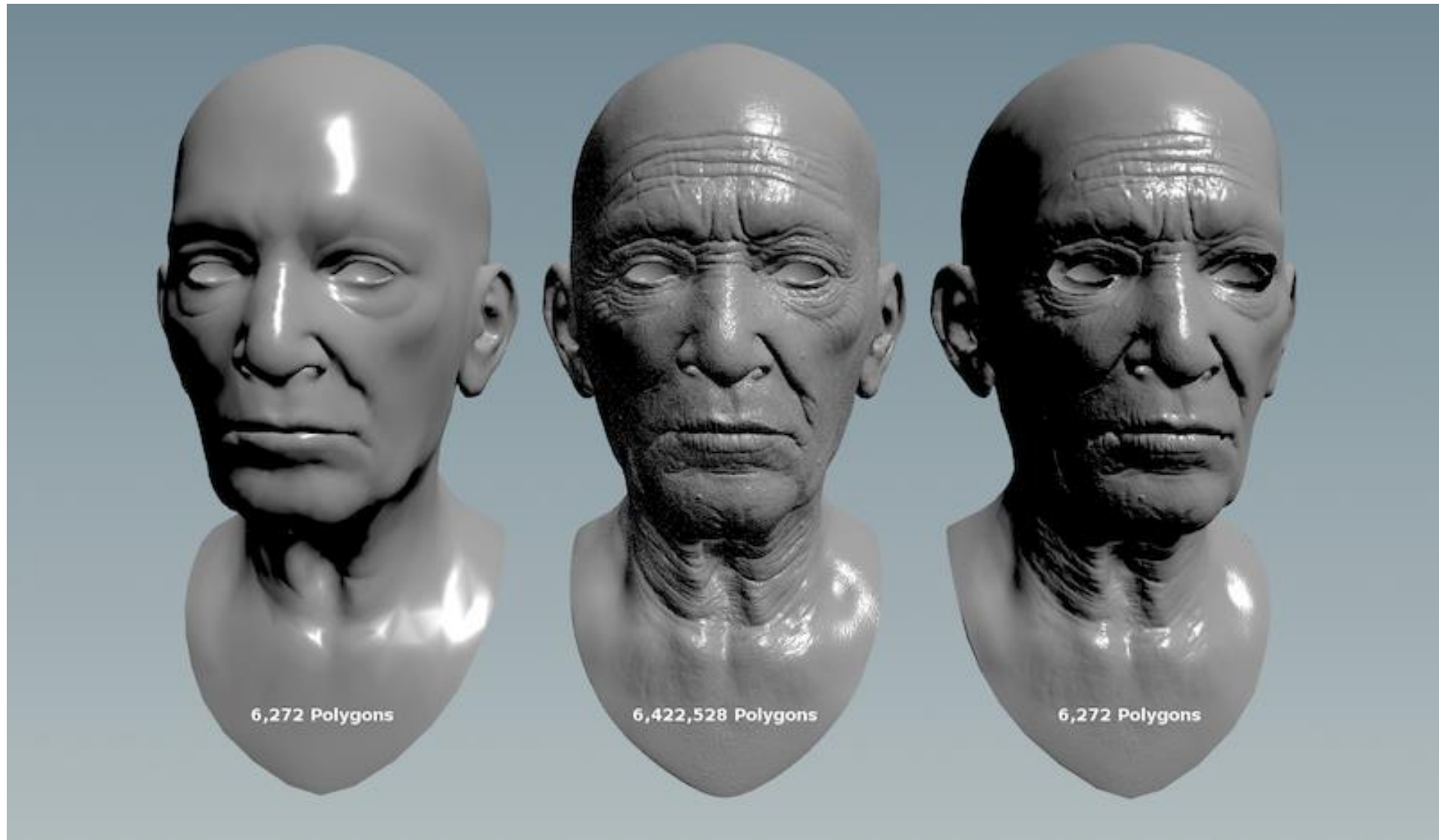
# Bump Maps in WebGL



## Bump Maps in WebGL

- The textures that we have encountered have affected color only. Whether they were image textures or procedural textures, their effect has been to vary the color on the surfaces to which they were applied.
- More generally, texture can refer to variation in any property. One example is **bump mapping**, where the property that is modified by the texture is the normal vector to the surface.
- A normal vector determines how light is reflected by the surface, which is a major visual clue to the direction that the surface faces.
- Modifying the normal vectors has the effect of modifying the apparent orientation of the surface. It can add the appearance of roughness or "bumps" to the surface.
- The effect can be visually similar to changing the positions of points on the surface, but with bump mapping, the change in appearance is achieved without actually changing the surface geometry.
- The alternative approach of modifying the actual geometry, which is called "**displacement mapping**", can give better results but requires a lot more computational and memory resources.

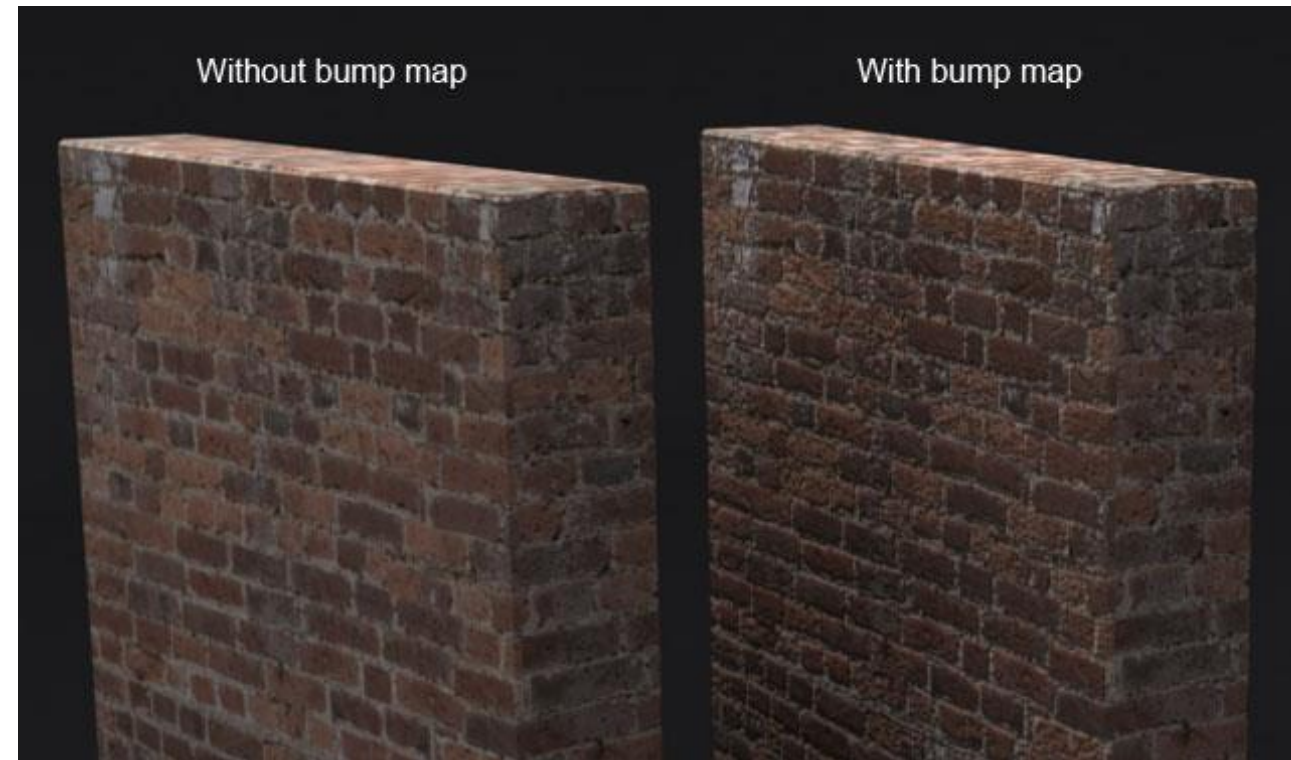
## Bump Maps in WebGL





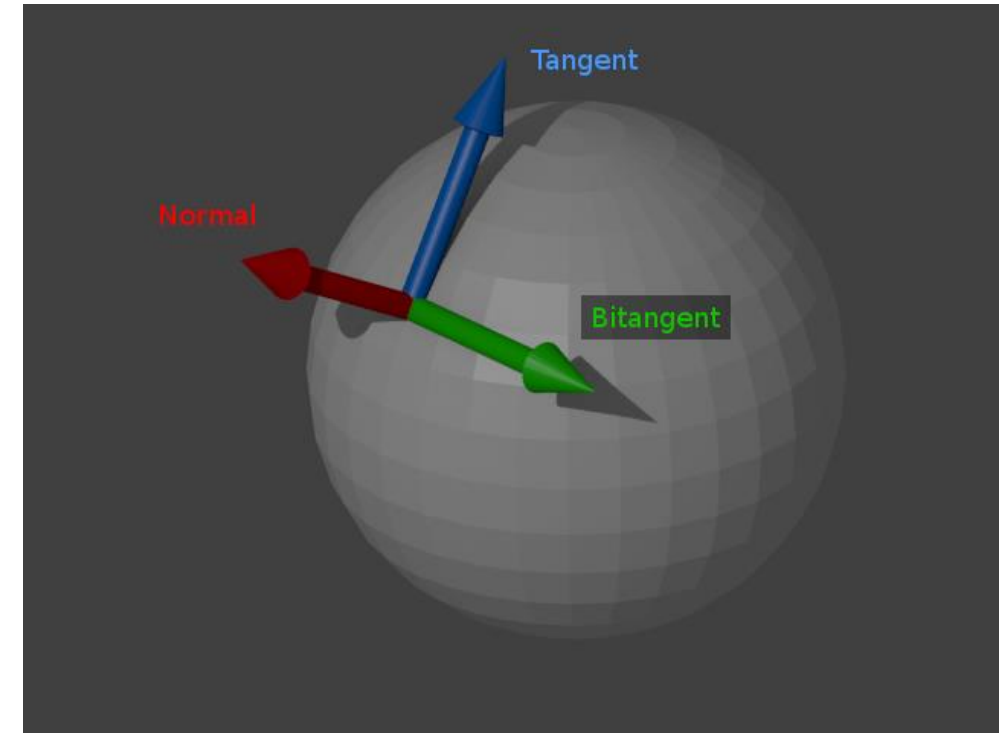
## Bump Maps in WebGL

- The typical way to do bump mapping is with a **height map**.
- A height map, is a grayscale image in which the variation in color is used to specify the amount by which points on the surface appear to be displaced.
- A height map is mapped to a surface in the same way as an image texture, using texture coordinates that are supplied as an attribute variable or generated computationally.
- Instead of being used to modify the color of a pixel, the color value from the height map is used to modify the normal vector that goes into the lighting equation that computes the color of the pixel.



## Bump Maps in WebGL

- To do bump mapping in 3D, we need to construct an orthonormal basis per vertex.
- This requires finding a tangent vector for each point on the surface. Tangent vectors will have to be provided, along with normal vectors, as part of the data for the surface.
- The tangent vector should point in the direction in which the s-coordinate in the texture coordinates is increasing.
- To account for variation in the height map, we need the bitangent vector. This second tangent vector is perpendicular both to the normal and to the first tangent vector.
- The bitangent vector points in the direction in which the t-coordinate is increasing.



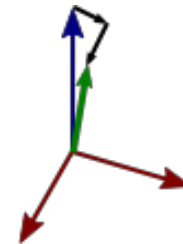
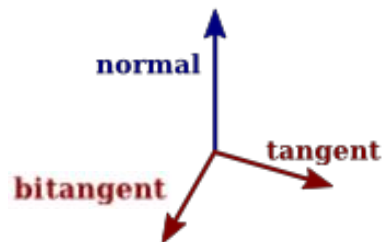
## Bump Maps in WebGL

- We can sample the height maps at two points, separated by a small difference in the s-coordinate. Let **a** be the difference between the two values; **a** represents the rate at which the height value is changing in the direction of the tangent vector.
- Now, sample the height map at two points separated by a small difference in the t-coordinate, and let **b** be the difference between the two values, so that **b** represents the rate at which the height value is changing in the direction of the bitangent vector.
- Let  $D = a * T + b * B$ , where **T** is the tangent vector and **B** is the bitangent vector. Then add **D** (or a multiple of **D**) to the original normal vector to produce the modified normal that will be used in the lighting equation.



A pixel in the height map (light gray) and two neighboring pixels. The change in color between the pixels defines the two vectors shown in white.

The tangent, normal, and binormal vectors at the pixel on the surface that corresponds to the pixel in the bumpmap texture.



The small black vectors are some multiple of the white vectors from the height map. They are added to the normal vector to produce the new normal vector, shown in green.

## Bump Maps in WebGL

- The bump map example on the blackboard demonstrates bump mapping.
- The bump mapping is implemented in the fragment shader in this example. The essential problem is how to modify the normal vector.
- The first three lines compute the normal, tangent, and bitangent unit vectors.
- The normal and tangent come from **varying** variables whose values are interpolated from **attribute** variables, which were in turn input to the shader program from the JavaScript side.
- The bitangent vector, which is perpendicular to both the normal and the tangent, is computed as the cross product of the normal and tangent.

```
vec3 normal = normalize(v_normal);
vec3 tangent = normalize(v_tangent);
vec3 bitangent = cross(normal,tangent);
```

## Bump Maps in WebGL

- The next four lines get the values of the height map at the pixel that corresponds to the surface point that is being processed and at two neighboring pixels.
- **bm0** is the height map value at the current pixel, whose coordinates in the texture are given by the texture coordinates, **v\_texCoords**.
- The value for **bm0** is the red color component from the **bumpmap** texture; since the texture is grayscale, all of its color components have the same value.
- **bmUp** is the value from the pixel above the current pixel in the texture; the coordinates are computed by adding  $1.0/\text{bumpmapSize.y}$  to the y-coordinate of the current pixel, where **bumpmapSize** is a **uniform** variable that gives the size of the texture image, in pixels.

```
float bm0, bmUp, bmRight; // Samples from the bumpmap at three texels.
bm0 = texture2D(bumpmap, v_texCoords).r;
bmUp = texture2D(bumpmap, v_texCoords + vec2(0.0, 1.0/bumpmapSize.y)).r;
bmRight = texture2D(bumpmap, v_texCoords + vec2(1.0/bumpmapSize.x, 0.0)).r;
```

## Bump Maps in WebGL

- Since texture coordinates in the image run from 0.0 to 1.0, the difference in the y-coordinates of the two pixels is `1.0/bumpmapSize.y`.
- Similarly, `bmRight` is the height map value for the pixel to the right of the current pixel in the `bumpmap` texture.

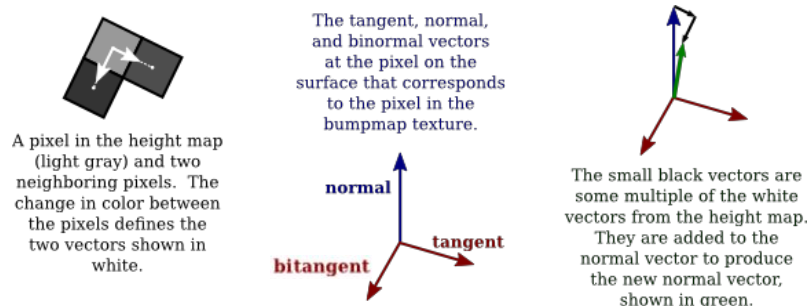
```
float bm0, bmUp, bmRight; // Samples from the bumpmap at three texels.
bm0 = texture2D(bumpmap, v_texCoords).r;
bmUp = texture2D(bumpmap, v_texCoords + vec2(0.0, 1.0/bumpmapSize.y)).r;
bmRight = texture2D(bumpmap, v_texCoords + vec2(1.0/bumpmapSize.x, 0.0)).r;
```

- Note that the minification filter for the `bumpmap` texture was set to `gl.NEAREST` because we need to read the actual value from the texture, not a value averaged from several pixels, as would be returned by the default minification filter.

## Bump Maps in WebGL

- The two vectors  $(bmRight - bm0) * tangent$  and  $(bmUp - bm0) * bitangent$  are the two white vectors in the previous illustration.
- Their sum is **bumpVector**. A multiple of that sum is added to the normal vector to give the modified normal vector. The multiplier, **bumpmapStrength**, is a **uniform** float variable.
- All of the calculations so far have been done in the object coordinate system. The resulting normal depends only on the original object coordinates, not on any transformation that has been applied.
- The normal vector still has to be transformed into eye coordinates before it can be used in the lighting equation. That transformation is done here.

```
vec3 bumpVector = (bmRight - bm0)*tangent + (bmUp - bm0)*bitangent;
normal += bumpmapStrength*bumpVector;
normal = normalize(normalMatrix*normal);
```





# Framebuffers



## Framebuffers

- The term "**framebuffer**" traditionally refers to the region of memory that holds the color data for the image displayed on a computer screen.
- In WebGL, a framebuffer is a data structure that organizes the memory resources that are needed to render an image.
- A WebGL graphics context has a default framebuffer, which is used for the image that appears on the screen.
- The default framebuffer is created by the call to `canvas.getContext()` that creates the graphics context. Its properties depend on the options that are passed to that function and cannot be changed after it is created.
- However, additional framebuffers can be created, with properties controlled by the WebGL program.
- They can be used for off-screen rendering, and they are required for certain advanced rendering algorithms.

## Framebuffers

- A framebuffer can use a **color buffer** to hold the color data for an image, a **depth buffer** to hold a depth value for each pixel, and a **stencil buffer** (for 3D shadows).
- The buffers are said to be "**attached**" to the framebuffer. For a non-default framebuffer, buffers can be attached and detached by the WebGL program at any time.
- A framebuffer doesn't need a full set of three buffers, but you need a color buffer, a depth buffer, or both to be able to use the framebuffer for rendering.
- If the depth test is not enabled when rendering to the framebuffer, then no depth buffer is needed.
- In WebGL 2.0, it is also possible to attach several color buffers to the same framebuffer, which can be useful for certain advanced algorithms and computational applications.
- The rendering functions `gl.drawArrays()` and `gl.drawElements()` affect the current framebuffer, which is initially the default framebuffer.
- The current framebuffer can be changed by calling:

```
gl.bindFramebuffer(gl.FRAMEBUFFER, framebufferObject);
```

## Framebuffers

Either `null` to select the default framebuffer, or a framebuffer created by the function `gl.createFramebuffer()`.

`gl.bindFramebuffer( gl.FRAMEBUFFER, framebufferObject );`

Always `gl.FRAMEBUFFER`.

## Framebuffers

- One use for a non-default framebuffer is to render directly into a texture. The basic idea is that we render a scene just like we usually do, but this time in a texture that we can reuse later.
- This technique, which is called **render-to-texture**, is used in the example render-to-texture on the Blackboard.
- Texture memory is normally allocated when an image is loaded into the texture using the function `gl.texImage2D()`.
- However, there is a version of `gl.texImage2D()` that can be used to allocate memory without loading an image into that memory. Here is an example, from the sample program:

```
texture = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, texture);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, 512, 512,
 0, gl.RGBA, gl.UNSIGNED_BYTE, null);
```

## Framebuffers

- It is the `null` parameter at the end of the last line that tells `gl.texImage2D()` to allocate new memory without loading existing image data to fill that memory. Instead, the new memory is filled with zeros.
- The first parameter to `gl.texImage2D()` is the texture target. The target is `gl.TEXTURE_2D` for 2D textures.
- The fourth and fifth parameters specify the height and width of the image; they should be powers of two.
- Note that the texture object must first be created and bound; `gl.texImage2D()` applies to the texture that is currently bound to the active texture unit.

```
texture = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, texture);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, 512, 512,
 0, gl.RGBA, gl.UNSIGNED_BYTE, null);
```

## Framebuffers

- To attach the texture to a framebuffer, we need to create a framebuffer object and make that object the current framebuffer by binding it:

```
framebuffer = gl.createFramebuffer();
gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer);
```

- Then the function `gl.framebufferTexture2D()` can be used to attach the texture to the framebuffer:

```
gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0,
Always gl.FRAMEBUFFER. gl.TEXTURE_2D, texture, 0);
```

Says a color buffer is being attached.

The texture object.

the same texture target that was used in `gl.texImage2D()`.

The mipmap level (usually zero).

## Framebuffers

- With this setup, we are now ready to bind the framebuffer and draw to the texture.
- After drawing the texture, we can call:

```
gl.bindFramebuffer(gl.FRAMEBUFFER, null);
```

to start drawing again to the default framebuffer. At that point, the texture is ready for use in subsequent rendering operations.

- The texture object can be bound to a texture unit, and a **sampler2D** variable can be used in the shader program to read from the texture.
- It makes sense to use different shader programs for drawing to the texture and drawing to the screen. Recall that the function **gl.useProgram()** is used to specify the shader program.
- In the example, the texture is animated. During the animation, a new image is drawn to the texture for each frame of the animation.
- The texture image is 2D, so the depth test is disabled while rendering it. This means that the framebuffer doesn't need a depth buffer.

# Framebuffers

```
function draw() {
 /* Draw the 2D image into a texture attached to a framebuffer. */
 gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer);
 gl.useProgram(prog_texture); // shader program for the texture

 gl.clearColor(1,1,1,1);
 gl.clear(gl.COLOR_BUFFER_BIT); // clear the texture to white

 gl.enable(gl.BLEND); // Use transparency while drawing 2D image.
 gl.disable(gl.DEPTH_TEST); // framebuffer doesn't even have a depth buffer!
 gl.viewport(0,0,512,512); // Viewport is not set automatically!
 .
 . // draw the texture image, which changes in each frame
 .
 gl.disable(gl.BLEND);
 /* Now draw the main scene, which is 3D, using the texture. */
 gl.bindFramebuffer(gl.FRAMEBUFFER, null); // Draw to default framebuffer.
 gl.useProgram(prog); // shader program for the on-screen image
 gl.clearColor(0,0,0,1);
 gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
 gl.enable(gl.DEPTH_TEST);
 gl.viewport(0,0,canvas.width,canvas.height); // Reset the viewport!
 .
 . // draw the scene
 .
}
```





**Thank you.**  
**DOUGLAS**COLLEGE

**DOUGLAS**COLLEGE

## References

- Interactive Computer Graphics, a Top-Down Approach with WebGL, 7<sup>th</sup> Edition – Edward Angel and Dave Shreiner, Pearson.
- David J. Eck, Introduction to Computer Graphics, Version 1.3, August 2021. Notes and demos from this source are licensed under a [Creative Commons Attribution-Noncommercial-ShareAlike 4.0 License](#). Formatting and style changes have been made to adapt the content for presentations.