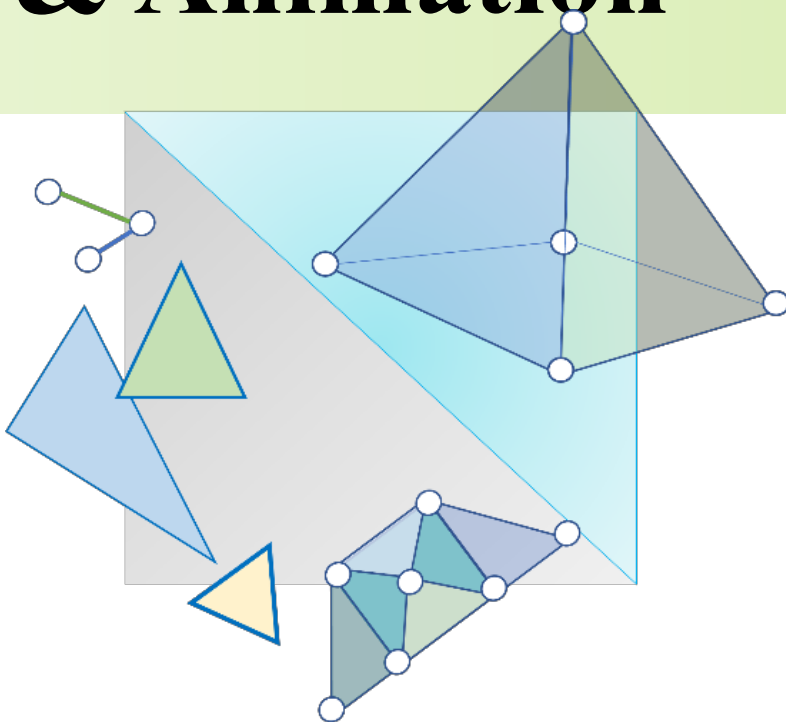
CITS3003 Graphics & Animation

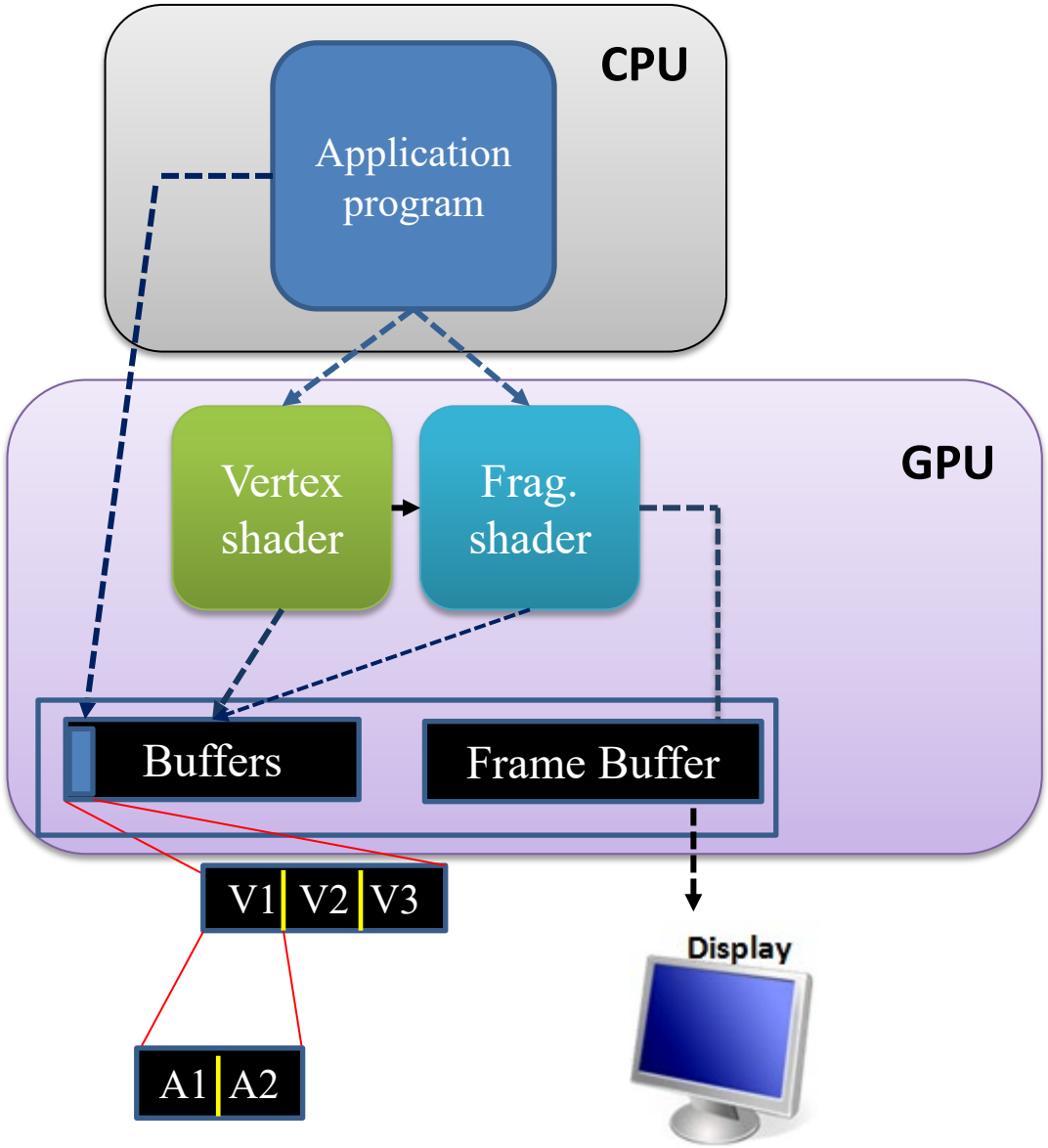


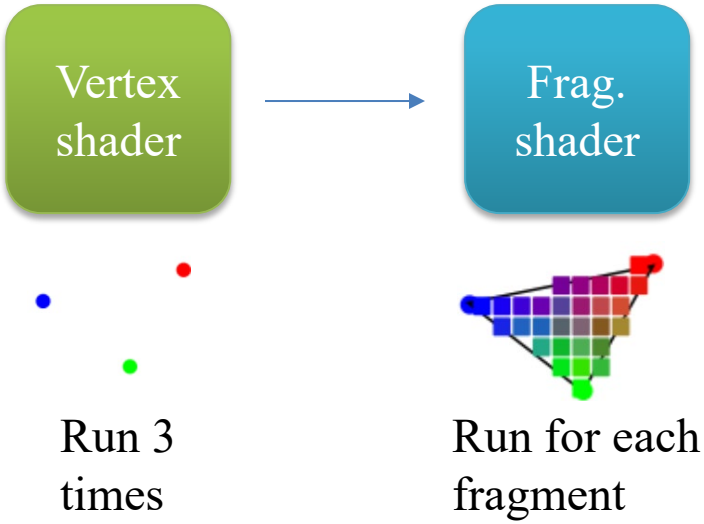
Lecture 4 : OpenGL: An Example Program

Content

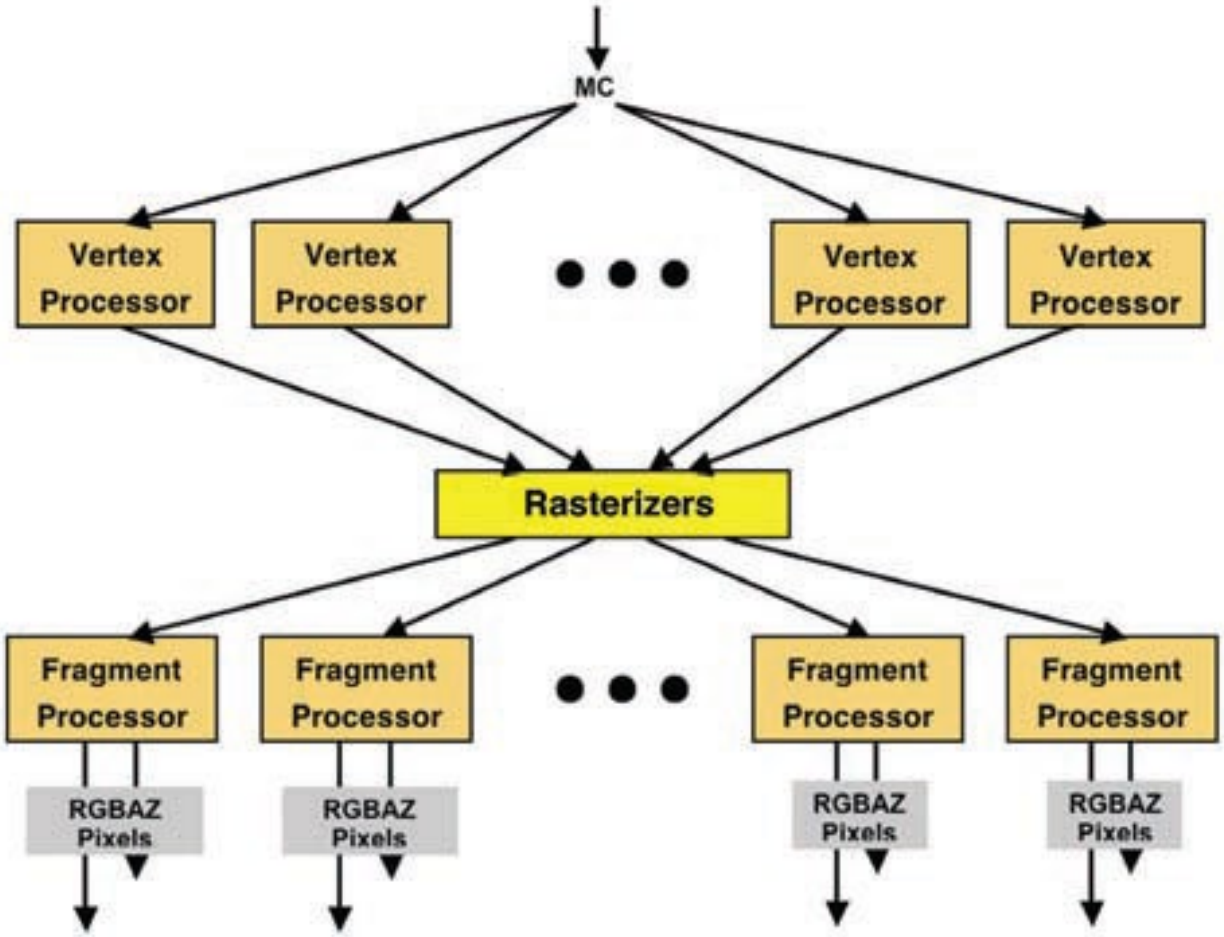
- Understand an OpenGL program
  - Initialization steps and program structure
  - GLUT functions
  - Vertex array objects and vertex buffer objects
- Simple viewing
  - Introduce the OpenGL camera, orthographic viewing, viewport, various coordinate systems, transformations

A Crude Visualization





A Crude Visualization



Abstracted parallelism in graphics processors

Image credits: Graphics Shaders (second edition) Bailey and Cunningham

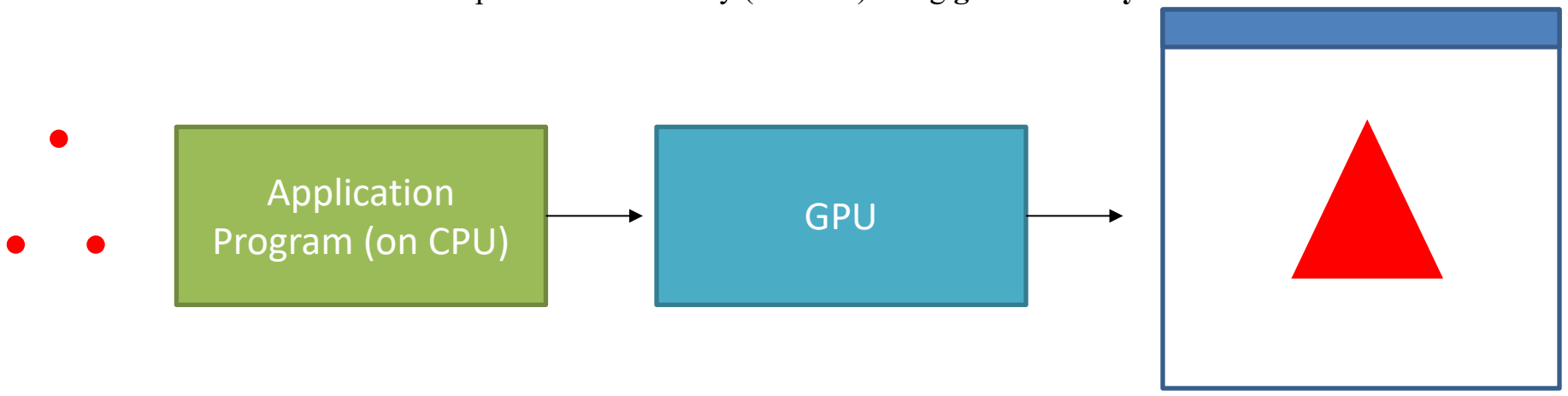
Example: Retained Mode Graphics

Task: Draw a triangle on white background

Simple Rendering Steps:

- Specify triangle corners (3 vertices)
- Store vertices into an array
- Create GPU Buffer for vertices
- Move array of 3 vertices from CPU to GPU
- Tell GPU to draw 3 points from an array (on GPU) using **glDrawArrays**

```
vec2 position[3];
    position[0] = vec2(-0.5, -0.5);
    position[1] = vec2(0.0, 0.5);
    position[2] = vec2(0.5, -0.5);
```



Rendered Triangle

OpenGL Program

Usually has 3 files:

- simple.cpp file: containing your main function
  - Does initialization, generates/loads geometry to be drawn

- Two shader files:
  - **Vertex shader**: functions to manipulate (e.g., move) vertices
  - **Fragment shader**: functions to manipulate pixels/fragments (e.g., change color)

A Simple Program (cont.)

- Most '.cpp' (simple.cpp in our case) files have a similar structure that consists of the following functions:
  - **main()**: creates the window, calls the **init()** function, specifies *callback* functions relevant to the application, enters event loop (last executable statement)
    - A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action
  - **init()**: defines the vertices, attributes, etc. of the objects to be rendered, specifies the shader programs,
  - **display():** this is a *callback* function that defines what to draw whenever the window is drawn/refreshed.

simple.cpp

```
#include 'Angel.h'
```

includes headers

```
void init() {
      // code to be inserted here
}
```

```
void mydisplay(){
      //this is a callback function
      //need to fill in this part
}
```

```
int main(int argc, char** argv){
      // create and open GLUT window;
      // call init();
      // register callback function;
      // wait in glutMainLoop for events;
}
```

simple.cpp

```
#include 'Angel.h'
```

includes headers

```
void init() {
      // code to be inserted here
}
```

```
void mydisplay(){
      //this is a callback function
      //need to fill in this part

}
```

```
int main(int argc, char** argv){
      // create and open GLUT window;
      // call init();
      // register callback function;
      // wait in glutMainLoop for events;
}
```

OpenGL programs are event driven

Event-driven program

- **Start at main( )**
- **Initialize**
- **Wait in infinite loop**
  - **Wait till defined event occurs**
  - **Event occurs => Take defined actions**

- The **main** function ends with the program entering an event loop

Display and Event Loop

- Note that the program specifies a *display* *callback* function named **mydisplay**

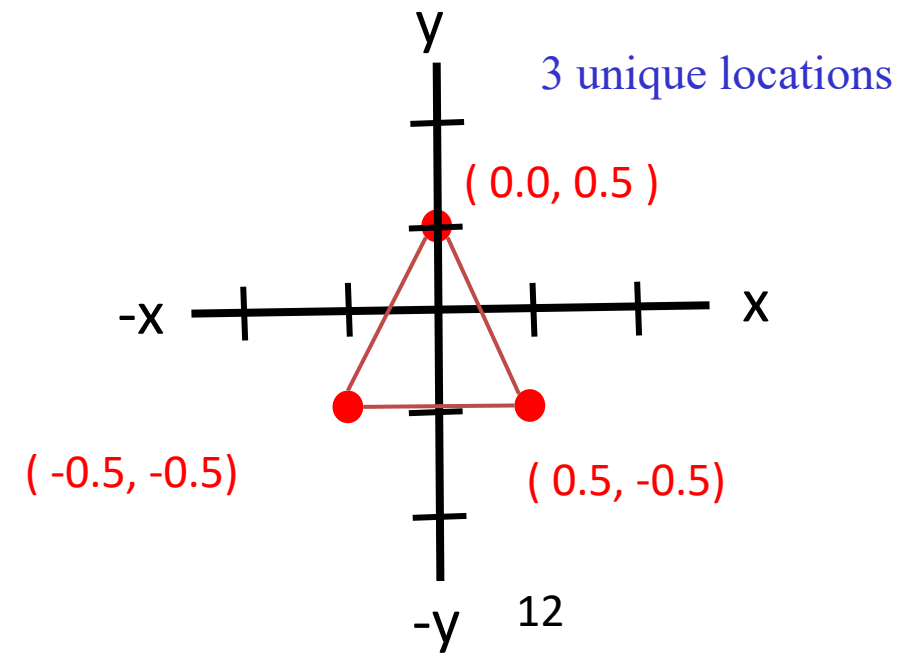
  - Every glut program must have a display callback
  - The display callback is executed whenever OpenGL decides the display must be refreshed, for example when the window is opened.

simple.cpp

```
#include "Angel.h"
```

includes `gl.h, glext.h, freeglut.h, vec.h, mat.h, …`

```
const int NumTriangles = 1;   // 1 triangles to be displayed
const int NumVertices  = 3 * NumTriangles;

vec2 points[NumVertices] = {
    vec2( -0.5, -0.5),
    vec2(  0.5, -0.5),
    vec2( 0.0, 0.5),
};
// generate vertices + store in an array
```



simple.cpp

```
int main(int argc, char **argv) {
   glutInit(&argc,argv);

   glutInitDisplayMode(GLUT_DOUBLE|GLUT_DEPTH);

   glutInitWindowSize(640,480);

   glutInitWindowPosition(100,150);

   glutInitContextVersion( 3, 2 );
   glutInitContextProfile( GLUT_CORE_PROFILE );

   glutCreateWindow("simple");
```

initializes the GLUT system and allows it to receive command line arguments

request “double buffering” & a “depth buffer”

specify window size and position

require OpenGL 3.2 Core profile

create a window with the title “simple”

simple.cpp

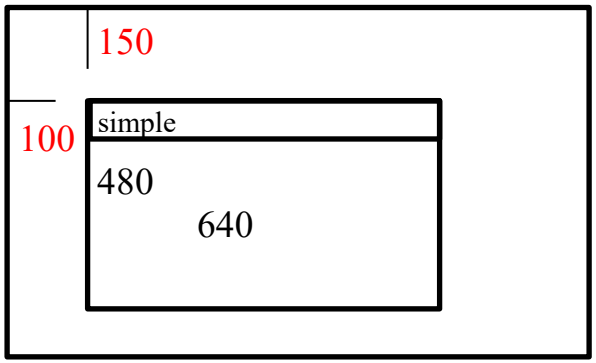
```
int main(int argc, char **argv) {
   glutInit(&argc,argv);

   glutInitDisplayMode(GLUT_DOUBLE|GLUT_DEPTH);
   glutInitWindowSize(640,480);
   glutInitWindowPosition(100,150);
   glutInitContextVersion( 3, 2 );
   glutInitContextProfile( GLUT_CORE_PROFILE );
   glutCreateWindow("simple");
```

initializes the GLUT system and allows it to receive command line arguments

request "double buffering" & a "depth buffer"

specify window size and position

create a window with the title "simple"



simple.cpp

```
int main(int argc, char **argv) {
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_DOUBLE|GLUT_DEPTH);

   glutInitWindowSize(640,480);
   glutInitWindowPosition(100,150);

   glutInitContextVersion( 3, 2 );
   glutInitContextProfile( GLUT_CORE_PROFILE );

   glutCreateWindow("simple");

   glewInit();

   init();

   glutDisplayFunc(mydisplay);

   glutMainLoop();
   return 0;
}
```

request "double buffering" & a "depth buffer"

specify window size and position

require OpenGL 3.2 Core profile

create a window with the title "simple"

set OpenGL state and initialize shaders

set display callback fn: **mydisplay** will be called when the window needs redrawing

enter event loop and wait for events

actually, never returns

GLUT functions

- **glutInit** initializes the GLUT system and allows it to receive command line arguments (always include this line)

- **glutInitDisplayMode** requests properties for the window (the *rendering context*)
  - RGBA colour (default) or indexed colour (rare now)
  - **Double buffering** (usually) or **Single buffering** (redraw flickers)
  - **Depth buffer** (usually in 3D) stores pixel depths to find closest surfaces
    - [usually with **glEnable(GL_DEPTH_TEST);**]
  - Others: **GLUT_ALPHA, …** generally for special additional window buffers
  - Properties are bitwise ORed together with | (vertical bar)

- **glutWindowSize** defines the window size in pixels
- **glutWindowPosition** positions the window (relative to top-left corner of display)

GLUT functions (cont.)

- **glutCreateWindow** creates a window
  - many functions need to be called prior to creating the window
  - similarly, many other functions can only be called afterwards

- **glutMainLoop** enters infinite event loop
  - never returns, but may exit

Callback Functions (Recall..):

- A callback function is a function which the library (GLUT) calls when it needs to know how to process events.
- Register callbacks for all events your program will react to
  - Example:
    - Declare function myMouse, to be called on mouse click
    - Register it: glutMouseFunc(myMouse)
- No registered callback = no action

GLUT functions (cont.)

Callback Registration

GLUT supports a number of callbacks to respond to events.

- **glutDisplayFunc** sets the display callback
- **glutKeyboardFunc** sets the keyboard callback
- **glutReshapeFunc** sets the window resize callback
- **glutTimerFunc** sets the timer callback
- **glutIdleFunc** sets the idle callback

OpenGl programs are event-driven:

Program responds to events such as:

- Mouse clicks
- Keyboard stroke
- Window resize

Initialization

All the initialization codes can be put inside an **init()** function. These include:

- Setting up the vertex array objects and vertex buffer objects
- Setting up vertex and fragment shaders
  - Read in the shaders
  - Compile them
  - Link them

Covered in detail in later lectures

- Clearing window's background and other OpenGL parameters

init()

```
  void init( void )
{
   // Create a vertex array object
   GLuint vao;
   glGenVertexArrays( 1, &vao );
   glBindVertexArray( vao );

 // Create and initialize a vertex buffer object
   GLuint buffer;
   glGenBuffers( 1, &buffer );
   glBindBuffer( GL_ARRAY_BUFFER, buffer );

   // Move the six points generated earlier to VBO
   glBufferData( GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW );


// Load shaders and use the resulting shader program
   GLuint program = InitShader( "vertex.glsl", "fragment.glsl" );
   glUseProgram( program );

// Initialize the vertex position attribute from the vertex shader
   GLuint vPos = glGetAttribLocation( program, "vPosition" );
   glEnableVertexAttribArray( vPos );
   glVertexAttribPointer( vPos, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0) );

// create black background
 glClearColor( 0.0, 0.0, 0.0, 0.0 ); /* black background */

}
```

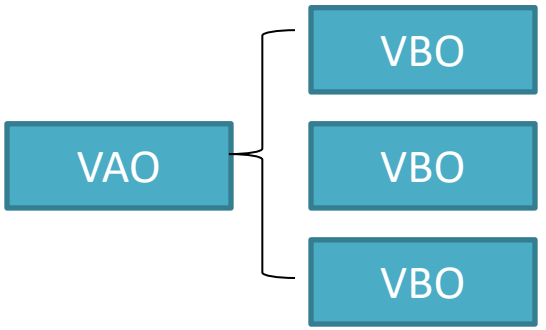
init()

Rendering from GPU memory significantly faster. Move data there

GPU memory for data called Vertex Buffer Objects (VBO)

Array of VBOs (called Vertex Array Object (VAO)) usually created

```
void init( void )
{
   // First Create a vertex array object
   GLuint vao;
   glGenVertexArrays( 1, &vao );
   glBindVertexArray( vao ); // make VAO active


 // Create and initialize a vertex buffer object
   GLuint buffer;
   glGenBuffers( 1, &buffer ); // create one buffer object

   glBindBuffer( GL_ARRAY_BUFFER, buffer );
```



Number of buffer objects to return

Data is array of values

init()

```
void init( void )
{
    // Create a vertex array object
    GLuint vao;
    glGenVertexArrays( 1, &vao );
    glBindVertexArray( vao );

 // Create and initialize a vertex buffer object
    GLuint buffer;
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );

    // Move the points generated earlier to VBO
    glBufferData( GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW );
```

buffer object data will not be changed. Specified once by application and used many times to draw

Data to be transferred to GPU memory (generated earlier)

Need to link names of vertex and fragment shaders to the main program

....

Vertex shader: functions to manipulate (e.g., move) vertices
Fragment shader: functions to manipulate pixels/fragments (e.g change color)

init()

```
void init( void )
{
    // Create a vertex array object
    GLuint vao;
    glGenVertexArrays( 1, &vao );
    glBindVertexArray( vao );

 // Create and initialize a vertex buffer object
    GLuint buffer;
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );

    // Move the six points generated earlier to VBO
    glBufferData( GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW );


// Load shaders and use the resulting shader program
    GLuint program = InitShader( "vertex.glsl", "fragment.glsl" );
    glUseProgram( program );

// Initialize the vertex position attribute from the vertex shader
    GLuint vPos = glGetAttribLocation( program, "vPosition" );
    glEnableVertexAttribArray( vPos );
    glVertexAttribPointer( vPos, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0) );

}
```

initShader() connects main program to the shader files

Want to make 3 vertices accessible as variable 'vPosition' in vertex shader

Location of vPosition

2 (x,y) floats per vertex

Data no normalized (0-1 range)

Data starts at offset from start of array

Reading, Compiling, and Linking Shaders

- The function InitShader defined in **InitShader.cpp** carries out the reading, compiling, and linking of the shaders.

```
//create a program object
GLuint program = InitShader("vshader.glsl", "fshader.glsl");
glUseProgram(program);
```

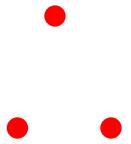
If there are errors in any of the GLSL files, the program will crash at this line.

Exercise: study **InitShader.cpp**.

Sending data to shaders using vertex buffer objects

Vertex Attributes

- Vertices can have many attributes
  - Position (1.0, 0.0, 0.1)
  - Color (e.g., red)
  - Texture Coordinates
  - Normal vector (x, y, z)

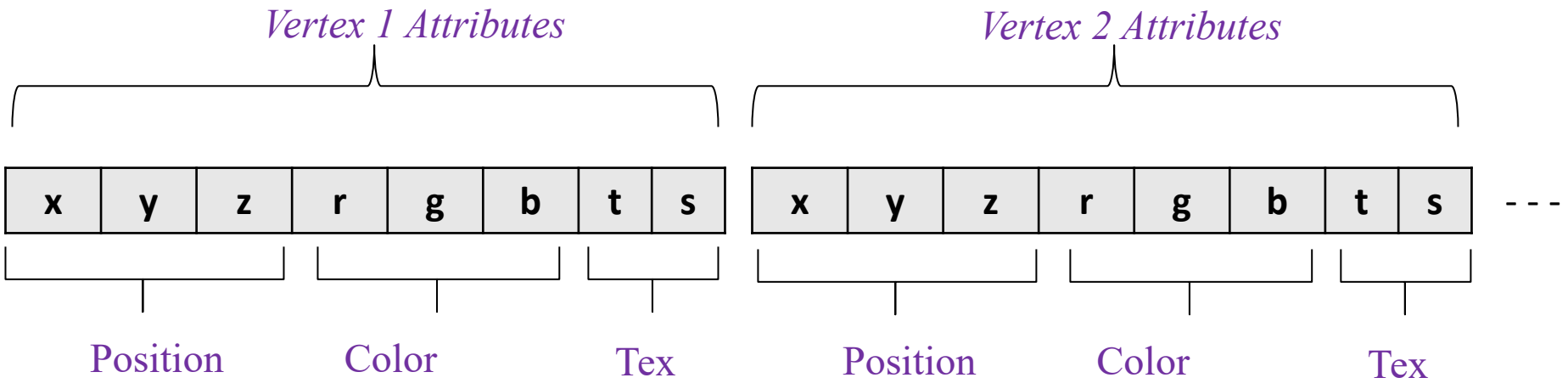


```
vec2 position[3] = {vec2(-0.5, -0.5),
                    vec2(0.0, 0.5),
                    vec2(0.0, -0.5)};

vec3 color[3] = {vec3(1.0, 0.0, 0.0),
                 vec3(1.0, 0.0, 0.0),
                 vec3(1.0, 0.0, 0.0)},
```

typedef in C: link

Vertex Attributes

- Vertices can have many attributes
  - Position
  - Color (e.g., red)
  - Texture Coordinates
  - Normal (x, y, z)



Vertex Array Objects

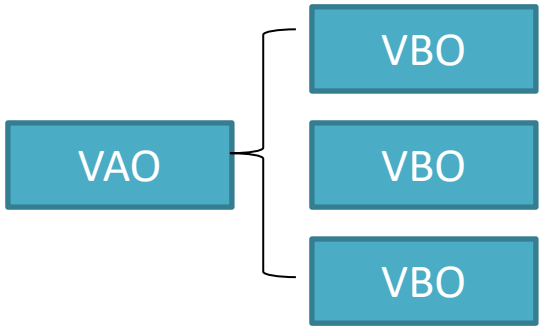
- Array of VBOs (called Vertex Array Object (VAO))
  - Example: vertex positions in VBO 1, color info in VBO 2, etc.,

- To define a vertex array object (VAO):
  1. Generate a vertex array object ID:

     ```
     GLuint vao;

     glGenVertexArrays(1, &vao);
     ```

  2. Bind the vertex array object ID

     ```
     glBindVertexArray(vao); // make VAO active
     //all subsequent vertex attribute and buffer operations will be associated with this VAO.
     ```



Vertex Array Object (cont.)

- Unfortunately, some openGL functions are not completely platform independent.
- On Linux/Windows:

```
GLuint abuffer;
glGenVertexArrays(1, &abuffer);
glBindVertexArray(abuffer);
```

- On Mac:

```
GLuint abuffer;
glGenVertexArraysAPPLE(1, &abuffer);
glBindVertexArrayAPPLE(abuffer);
```

Vertex Buffer Objects

- Vertex buffers objects (VBO) allow us to transfer large amounts of data to the GPU

- Need to create and bind the VBO then copy the vertices to the buffer:

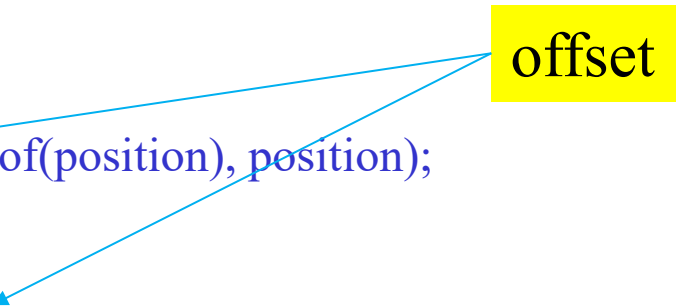
```
\\ create buffer object

Gluint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer); \\ make the VBO active
glBufferData(GL_ARRAY_BUFFER, sizeof(position), position, GL_STATIC_DRAW);
\\ move data to buffer object
```

Vertex Buffer Object (cont.)

How to update a portion of the data in an existing buffer object?

- **glBufferSubData** allows you to replace all or part of the data within the buffer

```
vec2 position[] = {...};
vec3 colour[] = {...};
//create a larger buffer to hold both points and colours

glBufferData(GL_ARRAY_BUFFER, sizeof(position) +
   sizeof(colour), NULL, GL_STATIC_DRAW);

//load data separately
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(position), position);


glBufferSubData(GL_ARRAY_BUFFER, sizeof(position), sizeof(colour), colour);
```



Vertex Buffer Object (cont.)

- Can also specify more than one buffer object, e.g., 2 buffer objects:

```
GLuint buffer[2];
glGenBuffers(2, buffer);

//do the binding and send the data for the 1st buffer
//to the GPU
glBindBuffer(GL_ARRAY_BUFFER, buffer[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(points), position);

//do the same for the 2nd buffer object
glBindBuffer(GL_ARRAY_BUFFER, buffer[1]);
glBufferData(GL_ARRAY_BUFFER,sizeof(colour), colour);
```

Passing vertex coordinates (variable **position**) and vertex colours (variable **colour**) to GPU using buffer[0] and buffer[1]

position could be an array of vec2, vec3, or vec4.
colour could be an array of vec3 or vec4.

Display Callback

- Once we get data to GPU, we can initiate the rendering with a simple display callback function:

```
void mydisplay()
{
      glClear(GL_COLOR_BUFFER_BIT);
      glDrawArrays(GL_TRIANGLES, 0, 3);

  // glFlush();    // Single buffering
      glutSwapBuffers();  // Double buffering
}
```

The *display* callback function is called every time the window needs to be repainted.

```
glDrawArrays(GL_POINTS, 0, N);
```

- **Render buffered data as points**
- **Starting index**
- **Number of points to be rendered**

- Prior to this, the vertex buffer objects should contain the vertex data.

slido

What are shaders in OpenGL?

ⓘ Start presenting to display the poll results on this slide.

slido

How often is a vertex shader run in the rendering pipeline?

ⓘ Start presenting to display the poll results on this slide.

Coordinate Frames

OpenGL commonly uses the following coordinate frames

- Object (Local) Coordinates
- World Coordinates
- Camera (View) Coordinates
- Clip Coordinates
- Window (or screen) Coordinates

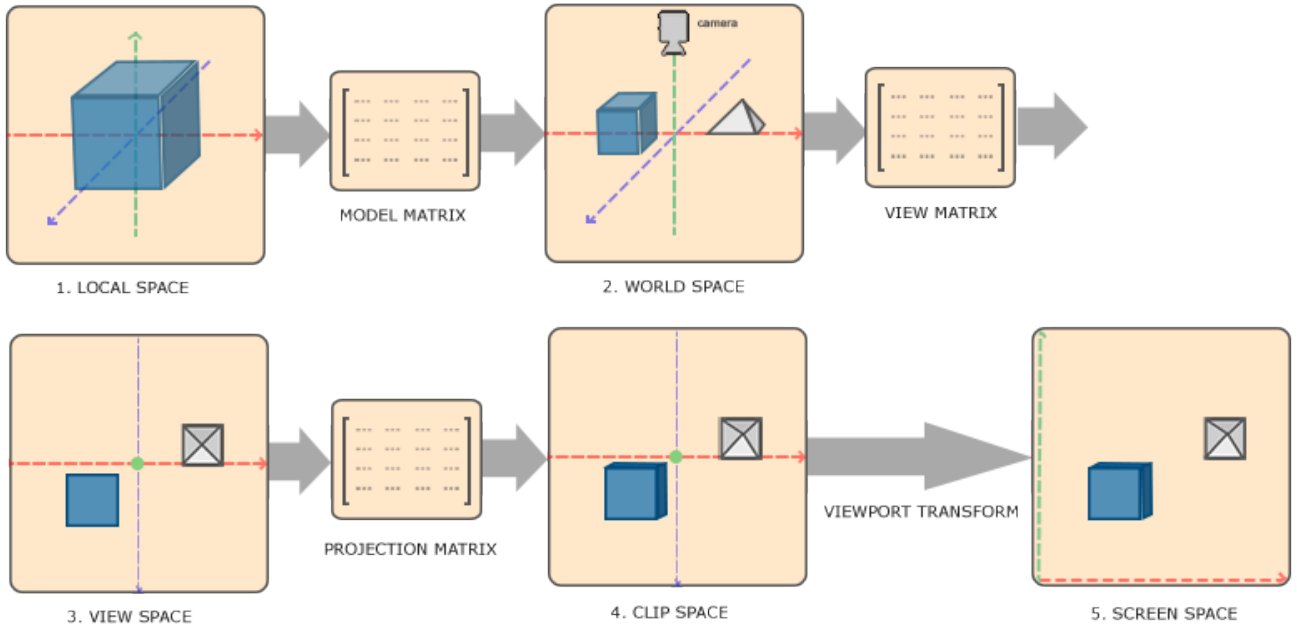
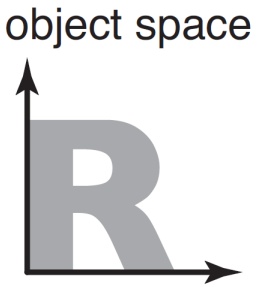


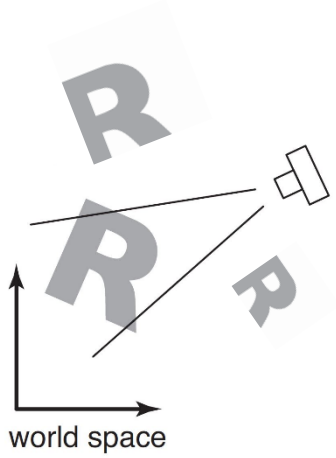
Figure Credits LInk

Coordinate Frames



Object Coordinates
This is the coordinate frame which is local to an object.

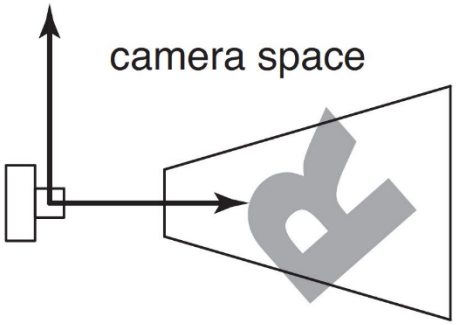
- We can define shapes of individual objects such as boxes, trees or mountains, within a separate coordinate frame for each object.



World Coordinates
The coordinates in which you build the complete scene are called world coordinates.

- Each virtual world may contain 100's of objects.
- The application program applies a sequence of transformations to orient and scale each object before placing them in the virtual world.

Coordinate Frames



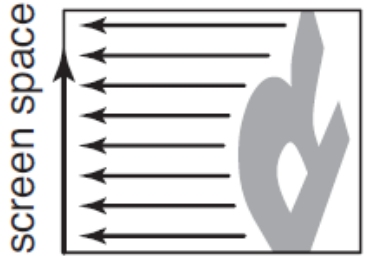
Camera (View) Coordinates
Camera coordinates are used to specify the position of objects relative to the camera (viewer's) position

- the camera/viewer is at the origin and looking into the negative z-direction. However, this can be altered in the program.

Coordinate Frames

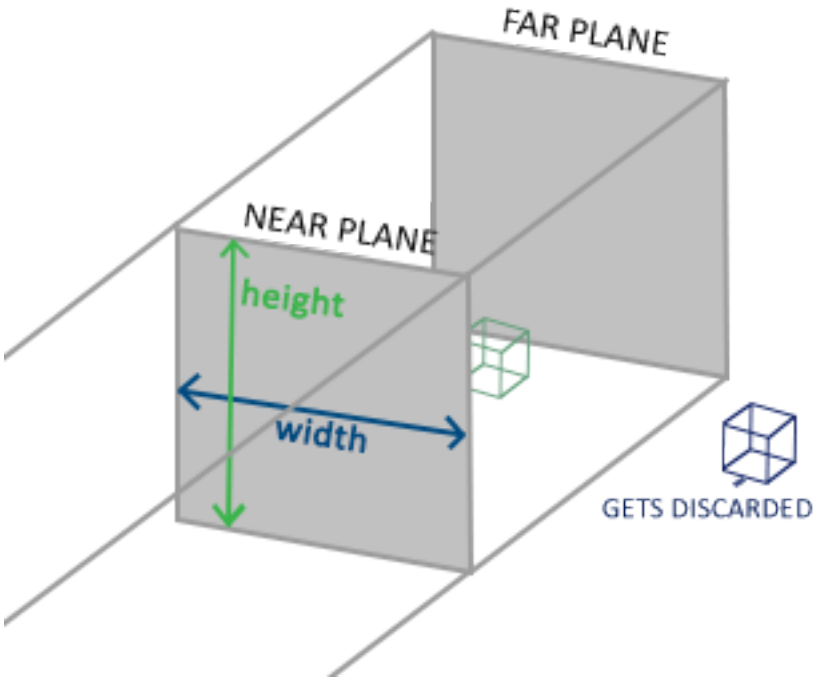
Clip Coordinates
Clip coordinates are used to specify the position of the objects relative to the clipping plane. Objects that are not inside the view volume are clipped out. Clip coordinates are used to operate in the clip space.

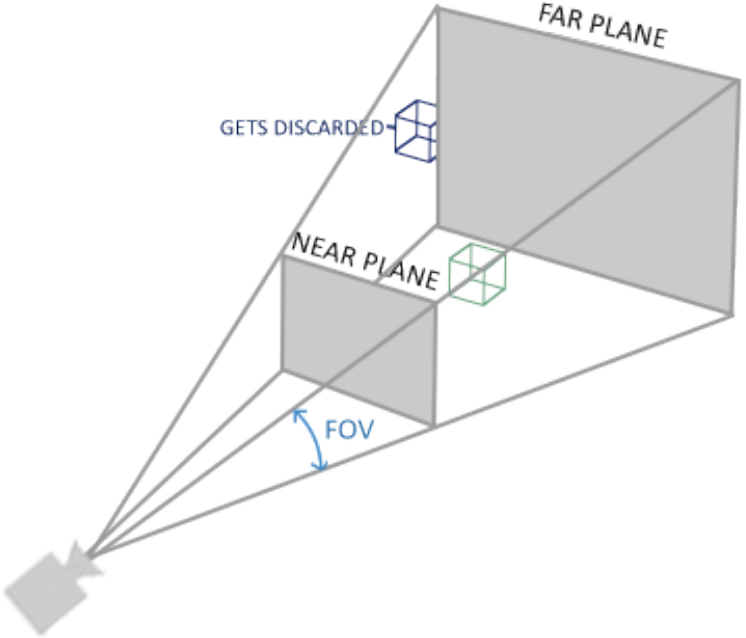
Window (or screen) Coordinates
This is the final coordinate system in which the output is rendered. Also known as device or screen coordinates. Window coordinates are expressed as a pair of numbers (x, y), which correspond to the position of an object on the screen.



Orthographic Vs Perspective Projection

We can either use Orthographic projection or perspective projection matrices to transform view coordinates to clip coordinates, where each form defines its own unique frustum.





Orthographic projection

- cube-like frustum box that defines the clipping space
- each vertex outside this box is clipped

Perspective projection

- a non-uniformly shaped frustum box defines the clipping space
- each vertex outside this box is clipped
- field of view and sets how large the viewspace is

Credits: link

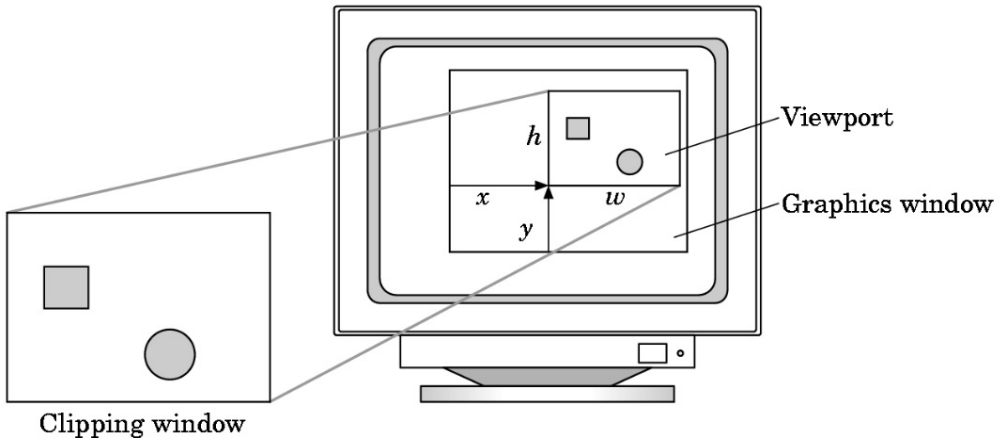
Viewports

- We do not have to use the entire window to render the scene, e.g., we can set the viewport like this:

glViewport(x,y,w,h)

- Values passed to this function should be in pixels (window coordinates)

- We can create multiple viewports in the same window

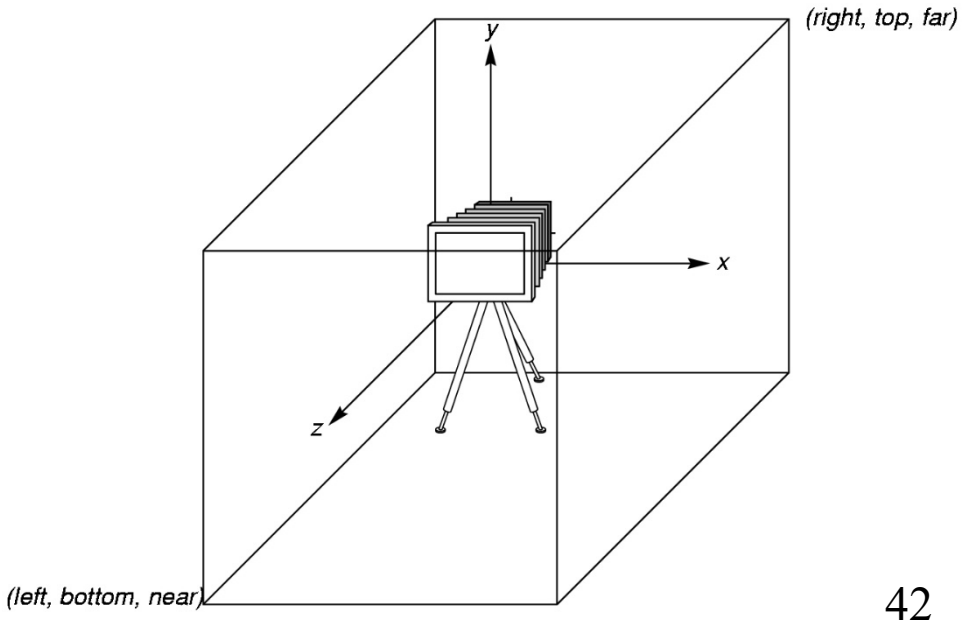


For example, if we want to draw two scenes, side-by-side, and that the drawing surface is 600-by-400 pixels. An outline for how to do that is very simple:

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

The OpenGL Camera

- OpenGL places a camera at the origin in the view coordinate space pointing in the negative *z* direction
- **The default viewing volume is a box centered at the origin with sides of length 2**



Further Reading

"Interactive Computer Graphics – A Top-Down Approach with Shader-Based OpenGL" by Edward Angel and Dave Shreiner, 6th Ed, 2012

- Secs. 2.1-2.2 The Sierpinski Gasket
- Sec. 2.6.1 The Orthographic View
- Sec 2.7 Control Functions
- Sec. 2.8 The Gasket Program
- Sec. 3.4 Frames in OpenGL (up to page 142)
- App. D.1 Initialization and Window Functions
- App. D.2 Vertex Array and Vertex Buffer Objects

Chapter#01, OpenGL 4.0 Shading Language Cookbook by David Wolff.

C Programming: What is a pointer? https://users.cs.cf.ac.uk/Dave.Marshall/C/node10.html