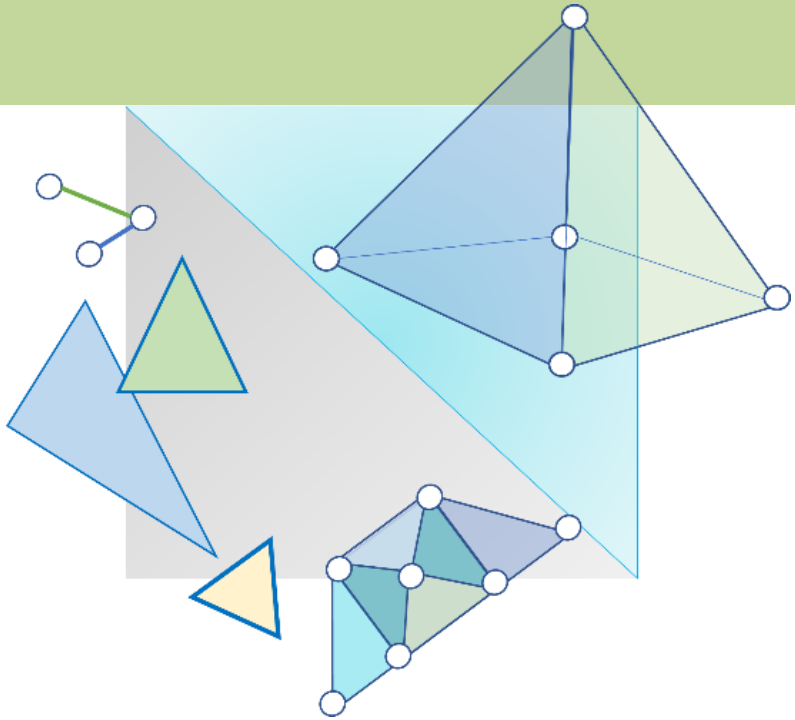
CITS3003 Graphics & Animation

Lecture 4 :
OpenGL: An Example Program

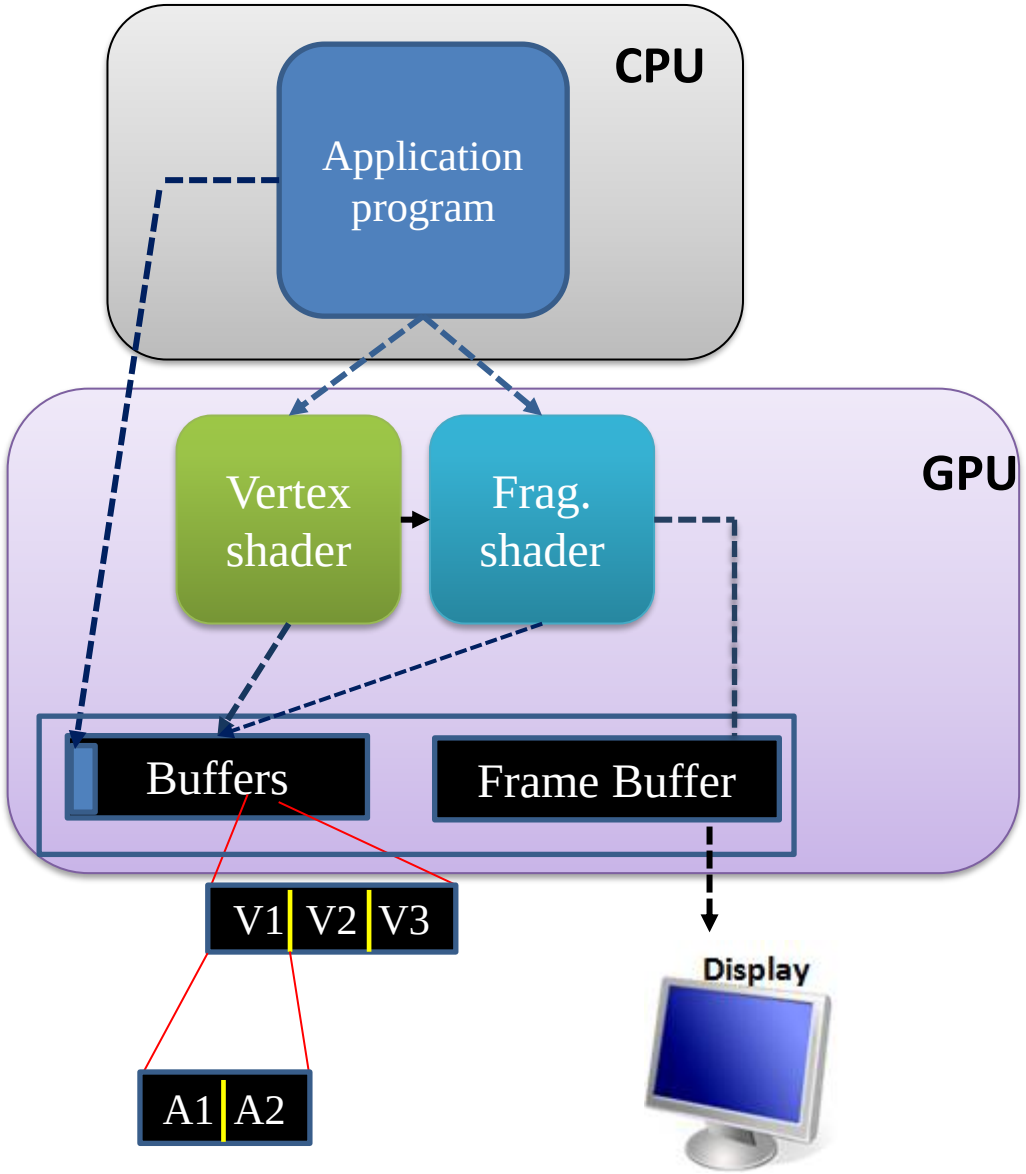


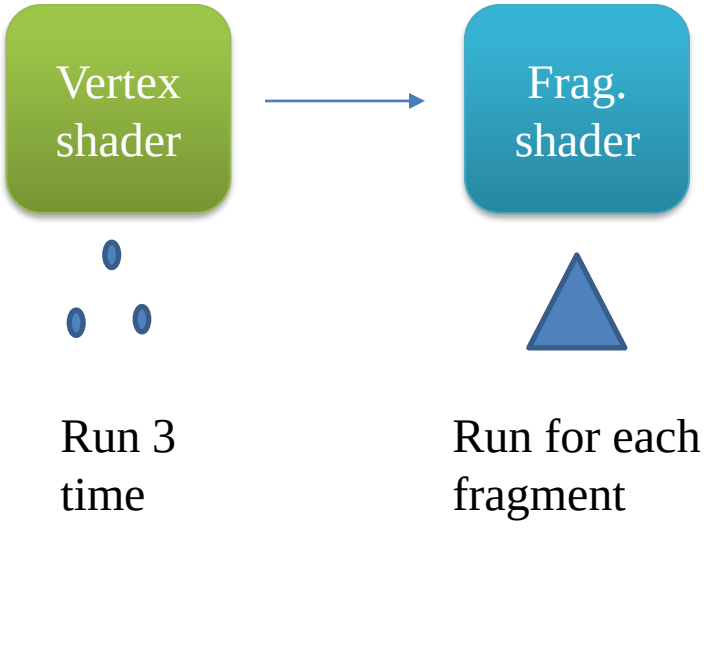
Content

- Understand an OpenGL program (revisiting simple.cpp from Lecture 2 and more)
  - 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





simple.cpp revisited

- **main()** – more complex than before; mostly calling GLUT functions
- **init()** – will incorporate color
  - **initShader()** – details of setting up shaders will be looked at in later lectures
- **display() -** callback
- Key issue is that we must form a data array to send to GPU and then render it

simple.cpp - A more complex version of main()

```
#include "Angel.h"

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");
```

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

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 - A more complex version of main()

```
#include "Angel.h"

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");
```

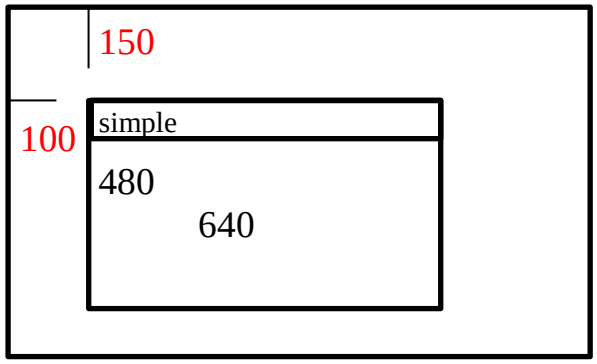
includes `gl.h, glext.h, freeglut.h, vec.h, mat.h, …`

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 - A more complex version of main()

```
#include "Angel.h"

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");

   glutDisplayFunc(mydisplay);

   glewInit();
   init();

   glutMainLoop();
   return 0;
}
```

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

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

set OpenGL state and initialize shaders

enter event loop

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 window with title "simple"
  - 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

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 only responds to events such as:

- Mouse clicks
- Keyboard stroke
- Window resize

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.

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

Initialization

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

- Setting up the vertex array objects and vertex buffer objects
- Clearing window's background and other OpenGL parameters
- Setting up vertex and fragment shaders
  - Read in the shaders
  - Compile them
  - Link them

Vertex Arrays

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

- A vertex array holds all of these data

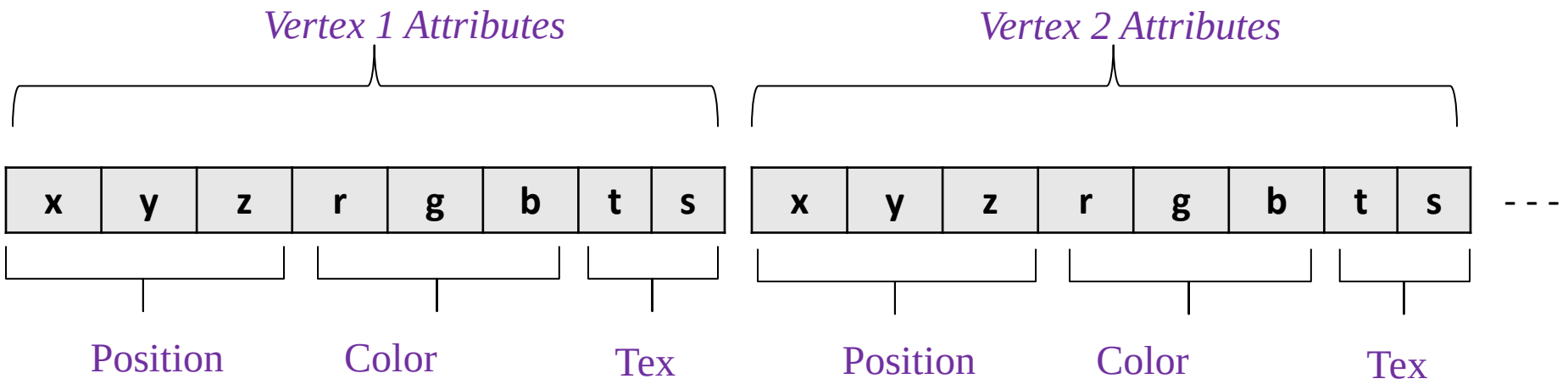
```
typedef vec2 point2  // point2 for (x,y) locations

point2 vertices[3] = {point2(-0.5, -0.5),
                      point2(0.0, 0.5),
                      point2(0.0, -0.5)};
```

typedef in C: link

Vertex Attributes

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



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:

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

- This is how data in the current vertex array is sent to GPU.

Vertex Buffer Object (cont.)

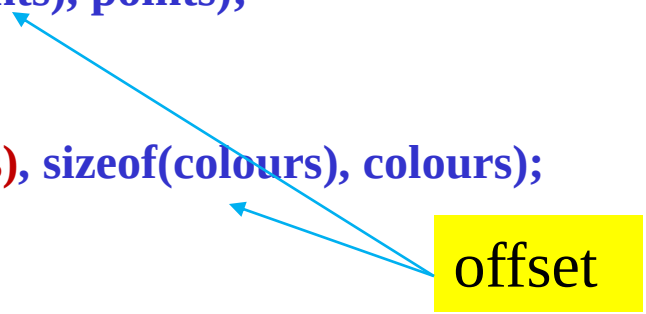
If we have a **points** array and a **colours** array, then we need to specify a larger buffer for the VBO and we need to call **glBufferSubData** for each array

```
//need a larger buffer as it needs to hold both points and colors

glBufferData(GL_ARRAY_BUFFER, sizeof(points) +
   sizeof(colours), NULL, GL_STATIC_DRAW);

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


glBufferSubData(GL_ARRAY_BUFFER, sizeof(points), sizeof(colours), colours);
```



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

points could be an array of vec2, vec3, or vec4.
colours could be an array of vec3 or vec4.

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

Vertex Array Objects

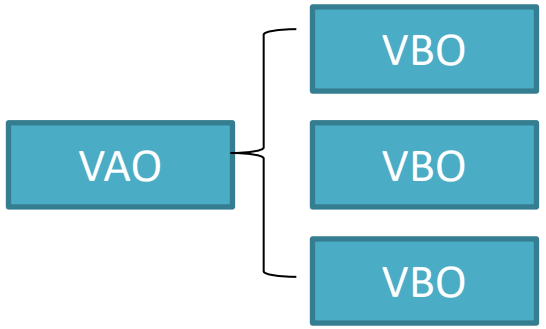
- Array of VBOs (called Vertex Array Object (VAO))
  - Example: vertex positions in VBO 1, color info in VBO 2, etc
- To define a **vertex array object** (**VAO**):
  1. Generate a vertex array object ID:

     GLuint vao;

     glGenVertexArrays(1, &vao);

  2. Bind the vertex array object ID

     glBindVertexArray(vao); // make VAO active



- At this point we have a current vertex array but no contents yet
- Use of glBindVertexArray lets us switch between VBOs

See later

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);
```

Reading, Compiling, and Linking Shaders

- In the **init()** function, we also read, compile, and link the vertex and fragment shaders to create a **program** object:

```
GLuint program = InitShader("vshader.glsl",
                            "fshader.glsl");
glUseProgram(program);
```

The function **InitShader** defined in **InitShader.cpp** carries out the reading, compiling, and linking of the shaders. If there are errors in any of the glsl file, the program will crash at this line.

Exercise: study **InitShader.cpp**.

Coordinate Frames

In OpenGL there are 6 coordinate frames specified in the pipeline:

1. Object (Local) Coordinates
2. World Coordinates
3. Camera (View) Coordinates
4. Clip Coordinates
5. Normalized Device Coordinates
6. Window (or screen) Coordinates

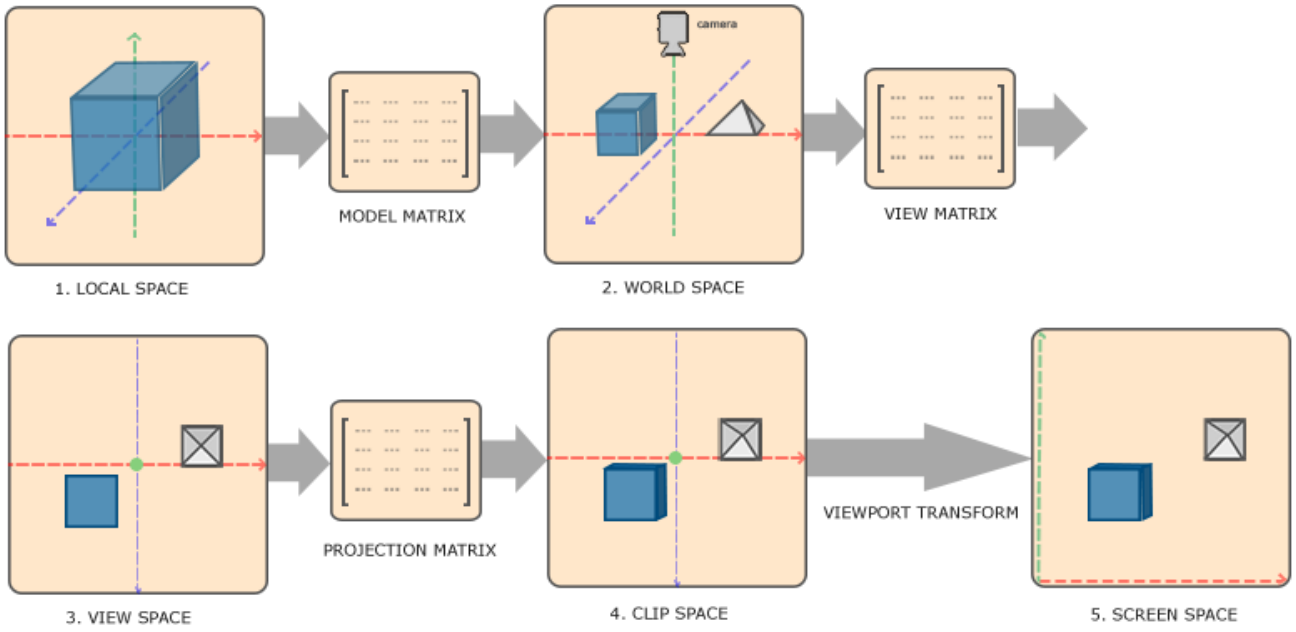


Figure Credits LInk

Coordinate Frames

In OpenGL there are 6 coordinate frames specified in the pipeline:

1. Object Coordinates
   The units of the **points** are determined in object, or model coordinates

2. World Coordinates
   The virtual worlds created in OpenGL are in 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.

3. Camera Coordinates
   All graphics systems use coordinate frames that are aligned with the camera coordinates i.e., the camera is at the origin and looking into the negative z-direction. However, this can be altered in the program.

4. Clip Coordinates
   Objects that are not inside the view volume are clipped out. Clip coordinates are used to operate in the clip space.

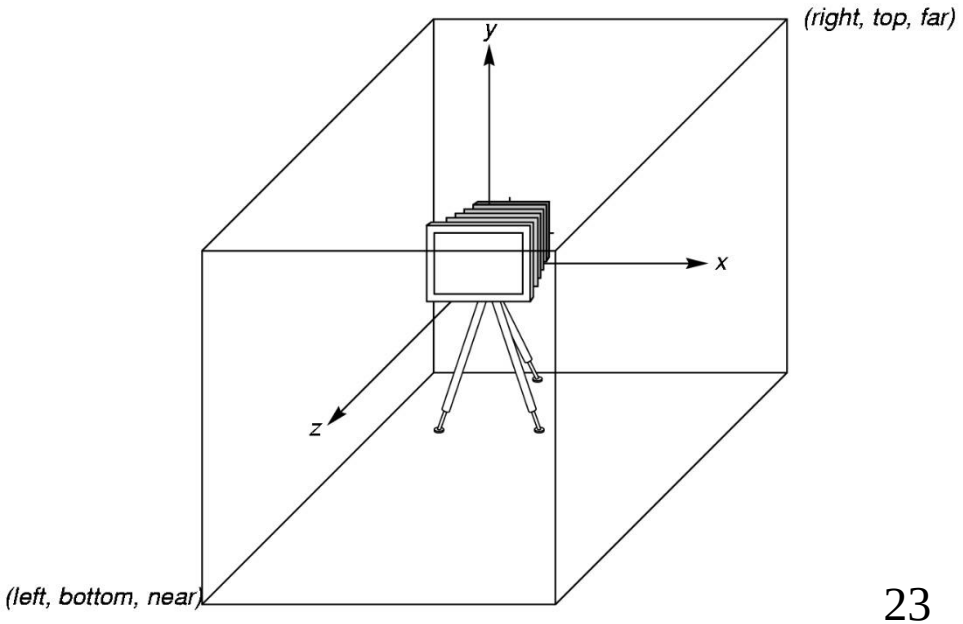
1. Normalized Device Coordinates
   Normalized device coordinate or NDC space is a screen independent display coordinate system; it encompasses a cube where the x, y, and z components range from −1 to 1.

1. Window (or screen) Coordinates
   Taking into account the viewport, creates 3D representation in window coordinates. Removing the depth value produces 2D window coordinates.

The OpenGL Camera

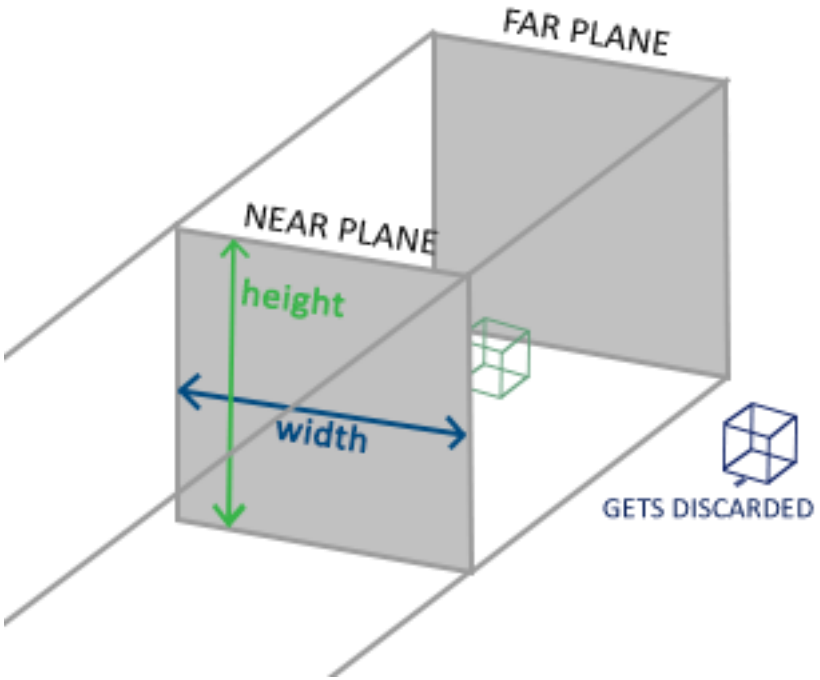
- OpenGL places a camera at the origin in the object coordinate space pointing in the negative *z* direction

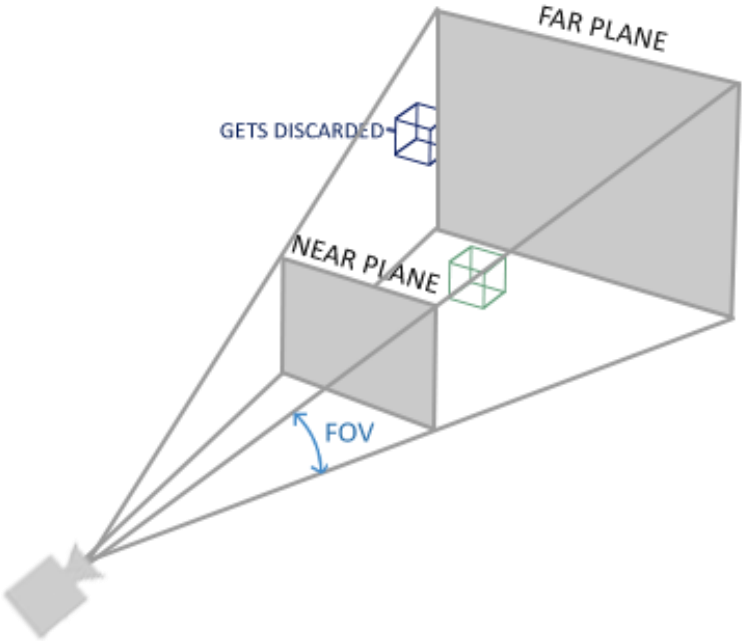
- **The default viewing volume is a box centered at the origin with sides of length 2**



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

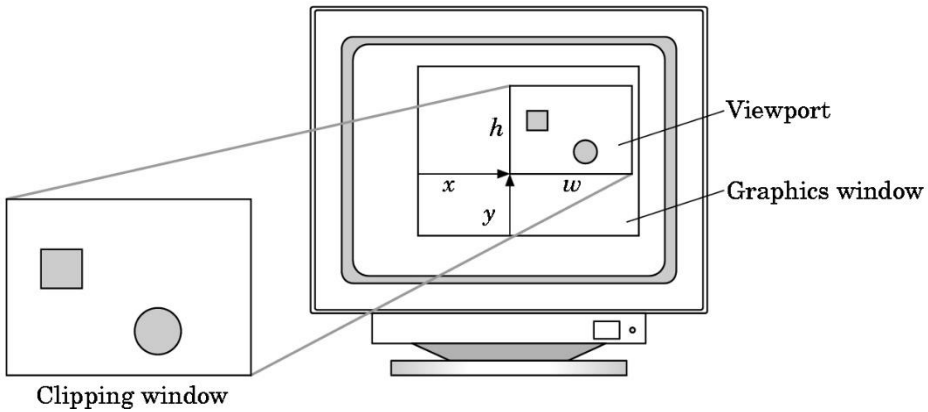
Credits: link

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)
- Can be used to maintain the aspect ratio of the scene
- We can create multiple viewports in the same window



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

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