

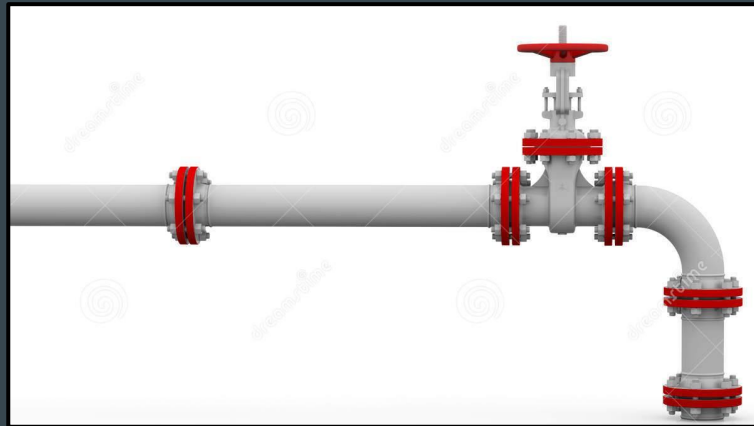
Intro to 3D Graphics

...

Carlos Fuentes

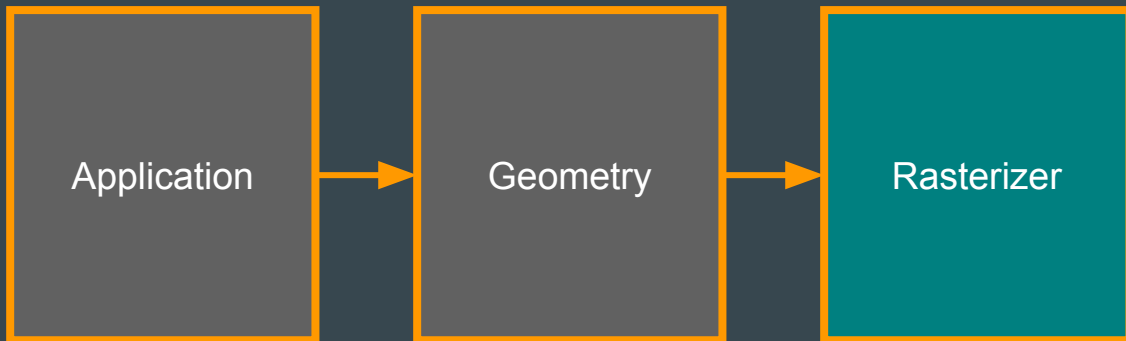
Pipeline

- The concept of a **pipeline** is widely used in video games
- Every process that follows stages that need to be completed is a pipeline
- The slowest step is called **bottleneck**
- Every step could contain a sub pipeline
- Some processes could be parallelized
- You have been using pipelines already



Real Time Rendering Pipeline

- Classical conceptual pipeline for a Graphics application was:



Real Time Rendering Pipeline

Application stage :

- Everything that happens before drawing
- Collision detection, input management, audio, animation, etc.
- All those process output geometry to be rendered later
- All this stage is done in the **CPU**

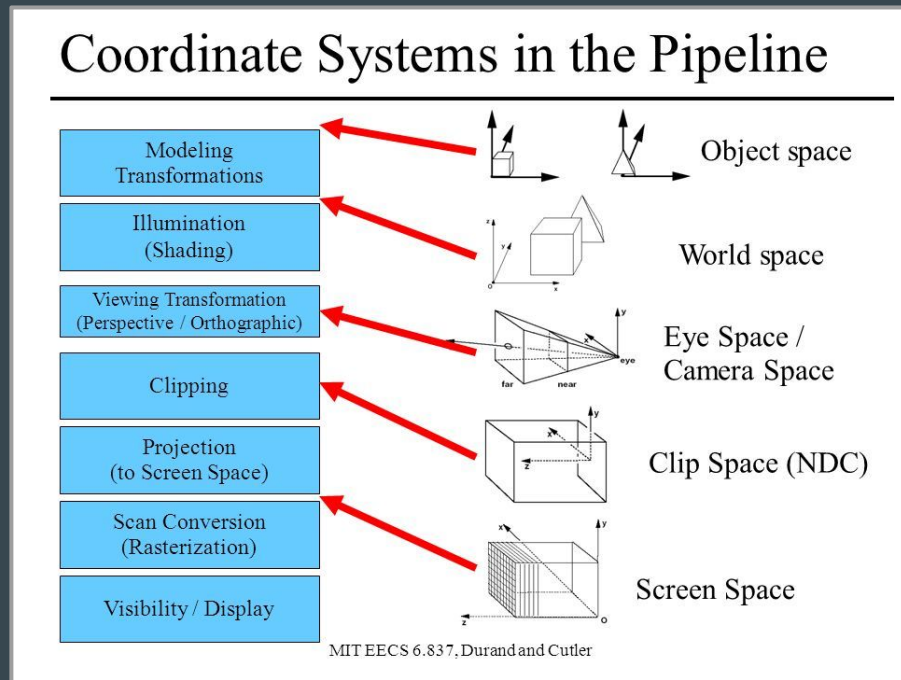
Real Time Rendering Pipeline

1. Geometry stage:

- Model & View Transform
- Lighting/Shading
- View/Projection
- Clipping
- Screen Mapping

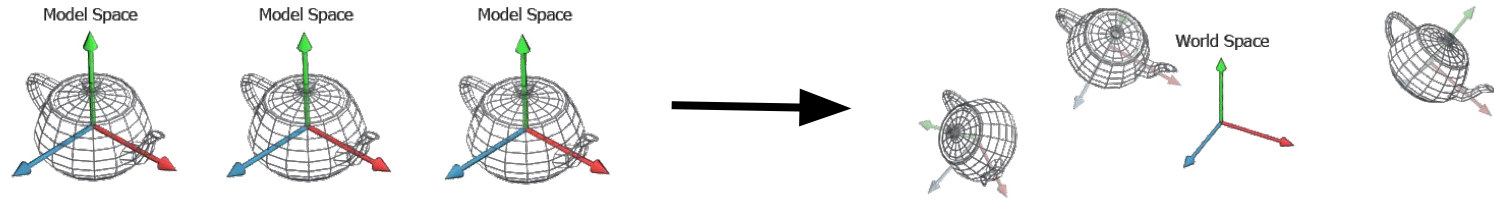
2. Rasterization Stage

- Rasterization
- Display



Real Time Rendering Pipeline

Model Transform:



Real Time Rendering Pipeline

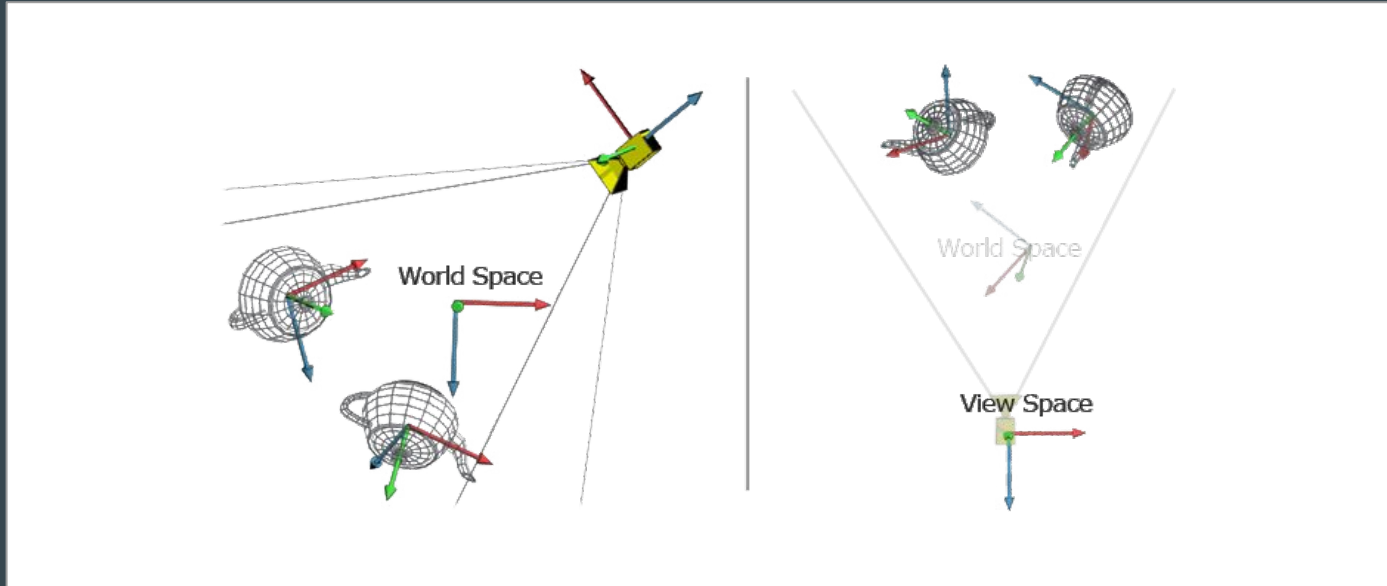
Lighting and Shading:

ambient + diffuse + specular = final result



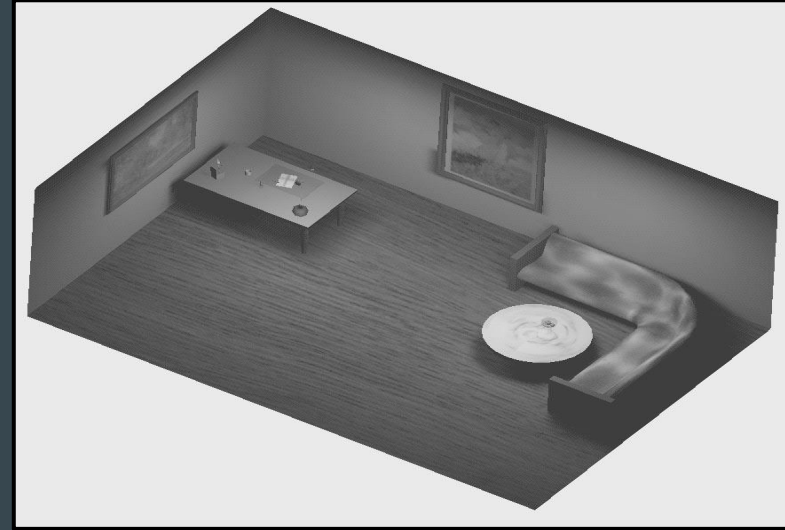
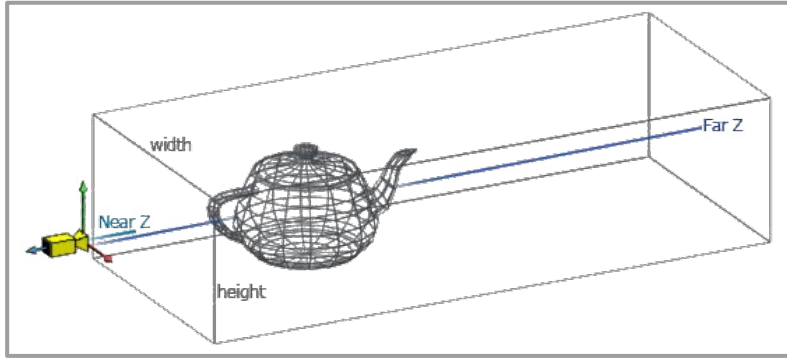
Real Time Rendering Pipeline

View Transform:



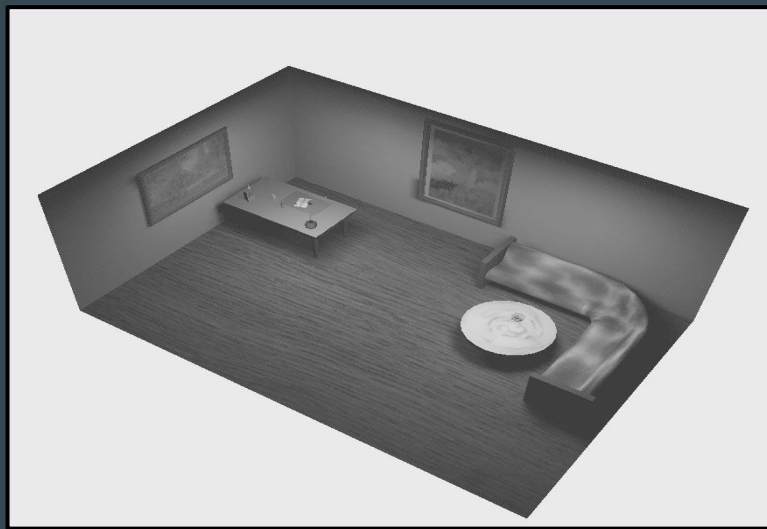
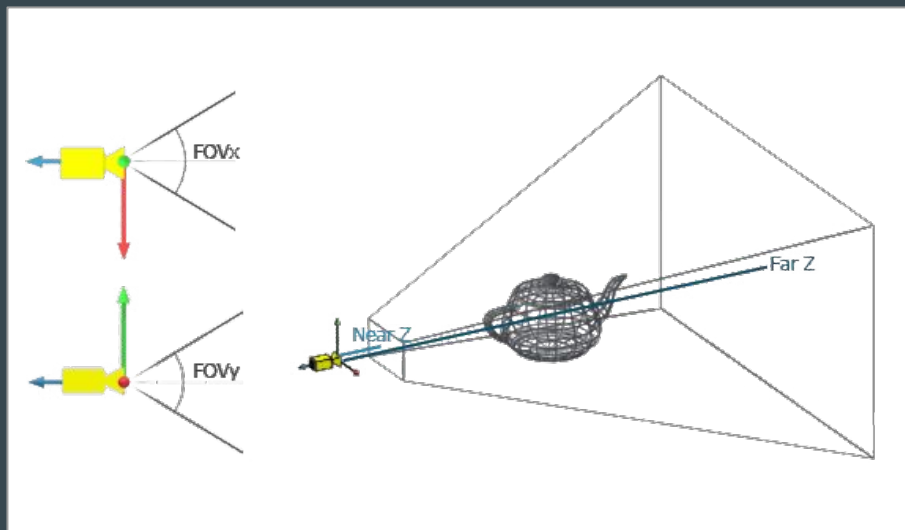
Real Time Rendering Pipeline

Orthographic Projection



Real Time Rendering Pipeline

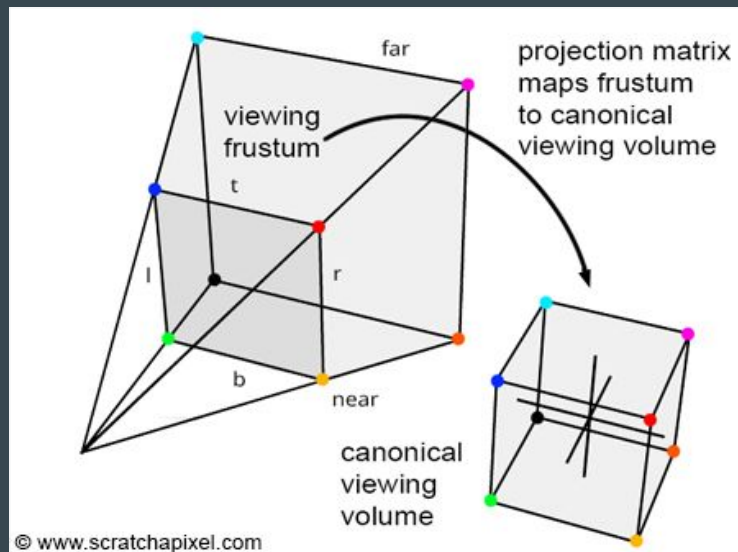
Perspective Projection



Real Time Rendering Pipeline

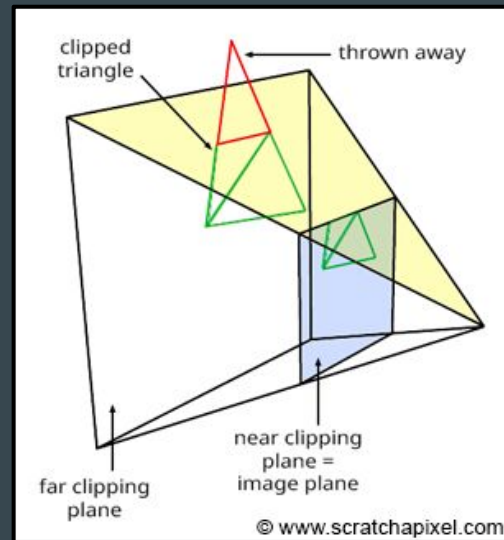
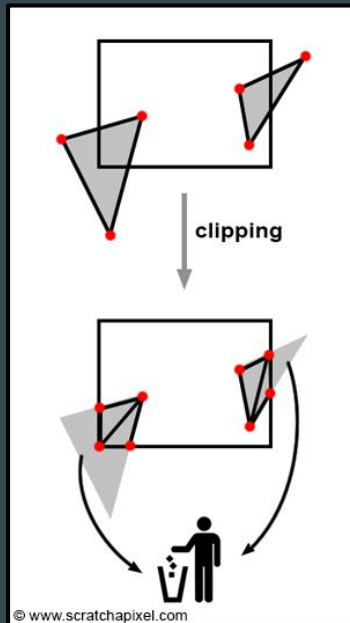
Clipping Space:

- Projection converts geometry to **Clipping Space**
- If we apply projection to camera frustum it's converted to a canonical view volume



Real Time Rendering Pipeline

- During clipping:
 - Primitives *totally inside* pass to next stage
 - Primitives *totally outside* are discarded
 - Primitives intersecting the cube are **clipped** (new vertices are created)



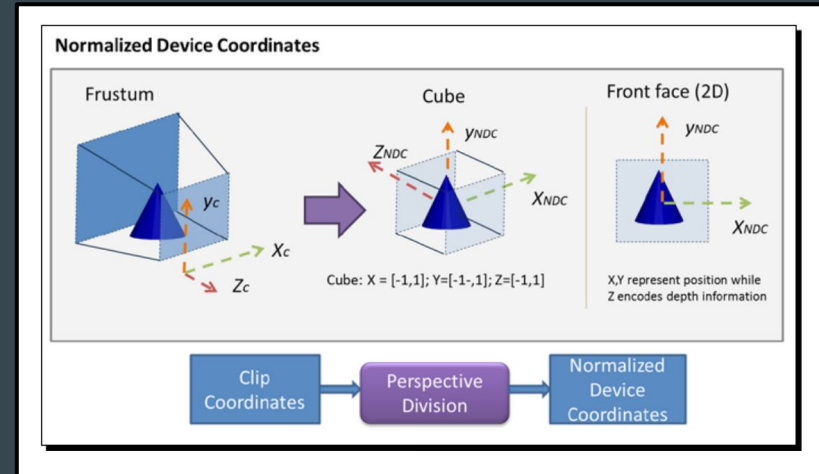
Real Time Rendering Pipeline

Normalized device coordinates:

- Dividing clipping space (positions results of projection) by w, converts them into

Normalized Device \Rightarrow Remaps canonical view volume to $[-1, -1, -1] \rightarrow [1, 1, 1]$

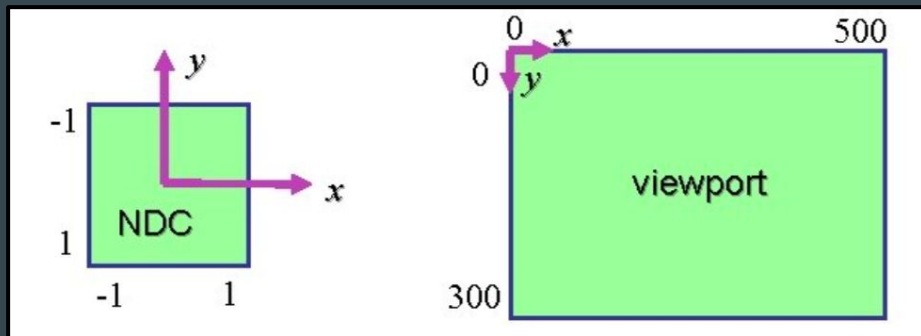
- These are the coordinates sent to next stage



Real Time Rendering Pipeline

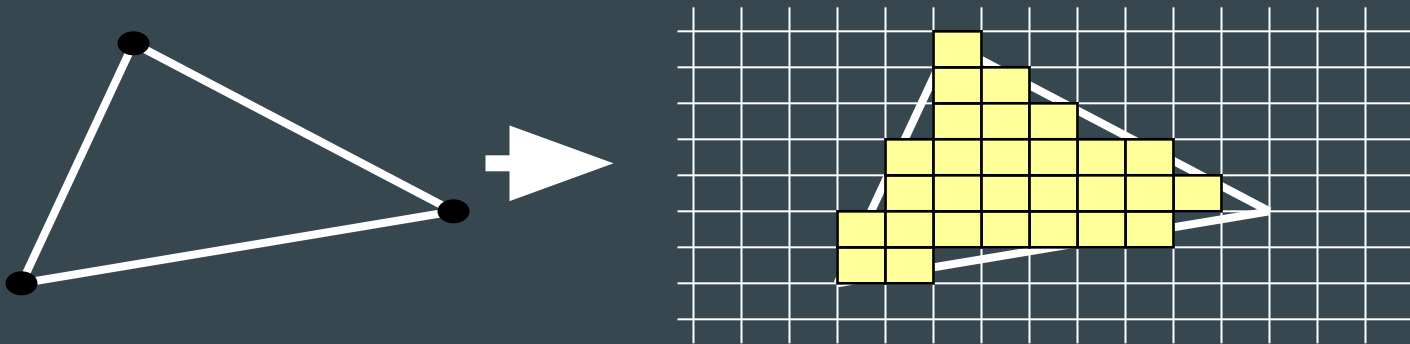
Screen Mapping:

- **Normalized Device Coordinates** x, y are mapped to screen/viewport.
- Z coordinate (depth value) remains the same.
- Screen space coordinates are sent to the rasterizer stage



Real Time Rendering Pipeline

- Turns geometry into pixels (discretization) to send to the screen
- Adds texturing and other pixel / fragment operations
- Uses z (depth) on each pixel to calculate visibility based on distance to camera



Main Graphics APIs

- **OpenGL**: generic and multiplatform (our choice)
- DirectX for Microsoft Platforms: proprietary
- Most games that execute on Microsoft platforms use DirectX
- Still, the differences are considered minor
- Now Vulkan vs DirectX 12 are becoming new standards
- Other APIs: WebGL, OpenGL ES, Metal

OpenGL Graphics

- OpenGL 1.0 (1992) direct mode was the main tool for drawing
- OpenGL 1.5 (2003) introduces *Vertex Buffer Objects*
- OpenGL 3.3 (2010) deprecated direct mode and forces *programmable shader pipeline* (tip: Shaders will be a personal feature)
- Latest is OpenGL 4.6 (2017) with all features supported also in DirectX 12
- Vulkan (2015) low overhead API to take over OpenGL

Documentation

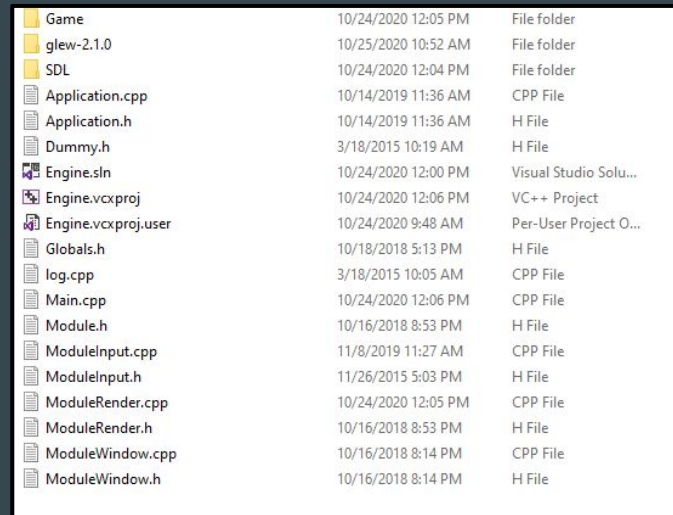
- [Real-Time Rendering](#) is the core book in the field
- [DirectX](#) site at Microsoft has several resources
- [OpenGL](#) site has articles and documentation to get you started
- Well explained pipeline for [OpenGL transformations](#)
- [Nice collection](#) of code about graphics
- OpenGL tutorials: [here](#), [here](#) and [here](#)
- But much more! Google is your friend :)

OpenGL Initialization

- OpenGL is big, with many versions and deprecated functionality
- Using OpenGL extensions (functions) is a disaster: It requires querying each function pointer before using it.
- We will use [glew lib](#) that helps to wrap each new OpenGL extension at application initialization.

Adding glew library

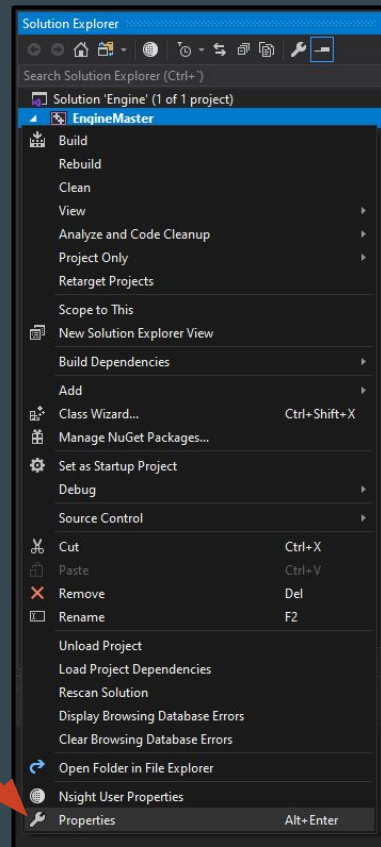
- Download binaries from <http://glew.sourceforge.net/> and add to your project directory
- Add glew to your projects directory
- Copy glew32.dll to your Game directory (will be your starting working directory)



Game	10/24/2020 12:05 PM	File folder
glew-2.1.0	10/25/2020 10:52 AM	File folder
SDL	10/24/2020 12:04 PM	File folder
Application.cpp	10/14/2019 11:36 AM	CPP File
Application.h	10/14/2019 11:36 AM	H File
Dummy.h	3/18/2015 10:19 AM	H File
Engine.sln	10/24/2020 12:00 PM	Visual Studio Solu...
Engine.vcxproj	10/24/2020 12:06 PM	VC++ Project
Engine.vcxproj.user	10/24/2020 9:48 AM	Per-User Project O...
Globals.h	10/18/2018 5:13 PM	H File
log.cpp	3/18/2015 10:05 AM	CPP File
Main.cpp	10/24/2020 12:06 PM	CPP File
Module.h	10/16/2018 8:53 PM	H File
ModuleInput.cpp	11/8/2019 11:27 AM	CPP File
ModuleInput.h	11/26/2015 5:03 PM	H File
ModuleRender.cpp	10/24/2020 12:05 PM	CPP File
ModuleRender.h	10/16/2018 8:53 PM	H File
ModuleWindow.cpp	10/16/2018 8:14 PM	CPP File
ModuleWindow.h	10/16/2018 8:14 PM	H File

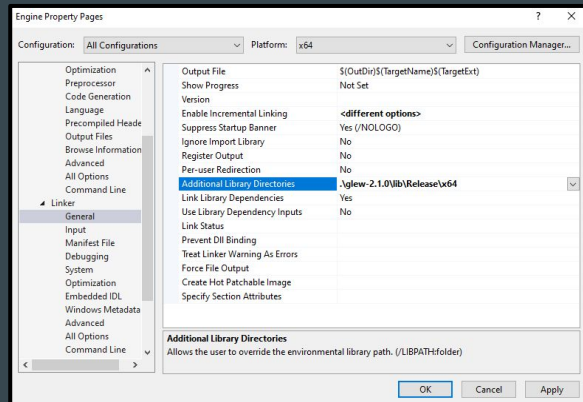
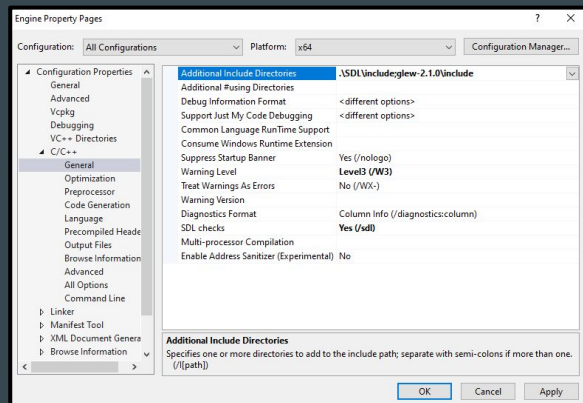
Adding glew library

Configure your Visual Studio project



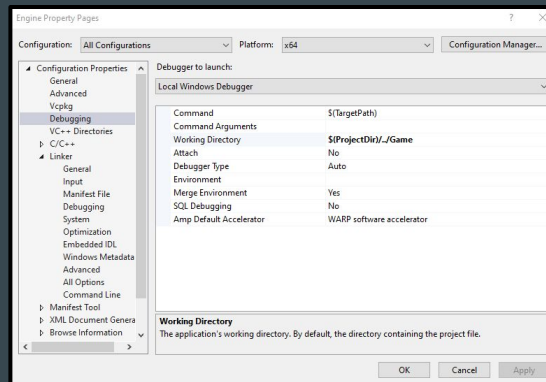
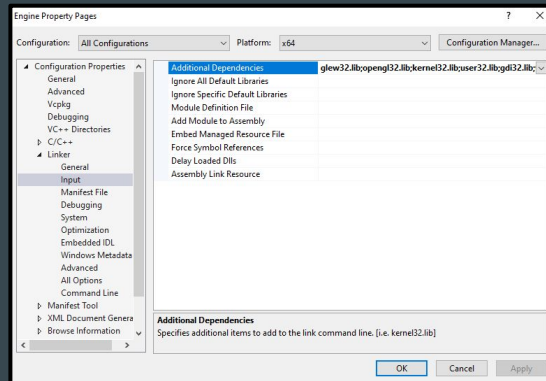
Adding glew library

- Add glew/include to your project additional include directories
- Add glew/lib/Release/x64 to your project additional library directories



Adding glew library

- Add glew32.lib and opengl32.lib to additional dependencies
- Set Game as your starting working directory



OpenGL Initialization

- For initializing OpenGL we must create a **Context**
- You must think a context as an object that holds all OpenGL.
- A Context contains all **states** associated with OpenGL: active framebuffer, active texture, active vertex buffer, etc.
- There are two types of context : **core** and **compatibility**. First removes deprecated functionality, second keeps it.

OpenGL Initialization

We will initialize OpenGL via SDL. Use *ModuleRender*

- Setup attributes with [SDL_GL_SetAttribute\(\)](#)

```
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 4); // desired version
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 6);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK, SDL_GL_CONTEXT_PROFILE_COMPATIBILITY);

SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1); // we want a double buffer
SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE, 24); // we want to have a depth buffer with 24 bits
SDL_GL_SetAttribute(SDL_GL_STENCIL_SIZE, 8); // we want to have a stencil buffer with 8 bits
```

- Init once with: [SDL_GL_CreateContext\(\)](#)

OpenGL Initialization

- Remember to create the SDL window with special flag:

`SDL_WINDOW_OPENGL` (*ModuleWindow*)

- After context creation, init the GLEW library (*ModuleRender*):

```
Glenum err = glewInit();  
// ... check for errors  
LOG("Using Glew %s", glewGetString(GLEW_VERSION));  
// Should be 2.0
```

OpenGL Initialization

- To detect our current hardware and driver capabilities we use glGetString()

```
LOG("Vendor: %s", glGetString(GL_VENDOR));  
LOG("Renderer: %s", glGetString(GL_RENDERER));  
LOG("OpenGL version supported %s", glGetString(GL_VERSION));  
LOG("GLSL: %s\n", glGetString (GL_SHADING_LANGUAGE_VERSION));
```

- Initialize some OpenGL global states (check documentation)

```
glEnable(GL_DEPTH_TEST);    // Enable depth test  
glEnable(GL_CULL_FACE);    // Enable cull backward faces  
glFrontFace(GL_CCW);       // Front faces will be counter clockwise
```

OpenGL Frame Init

1. For clearing the screen, on ModuleRender PreUpdate function:
 - a. Setup glViewport to 0, 0, window_width, window_height if window is resized:
 - i. Use SDL_GetWindowSize
 - b. glClearColor(0.1f, 0.1f, 0.1f, 1.0f);
 - c. glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
2. Swap frame buffer on ModuleRender PostUpdate function:
 - a. Use SDL_GL_SwapWindow
3. Remove context at CleanUp function using SDL_GL_DeleteContext()

OpenGL Destruction

