
LECTURE 1
Introduction

Introduction

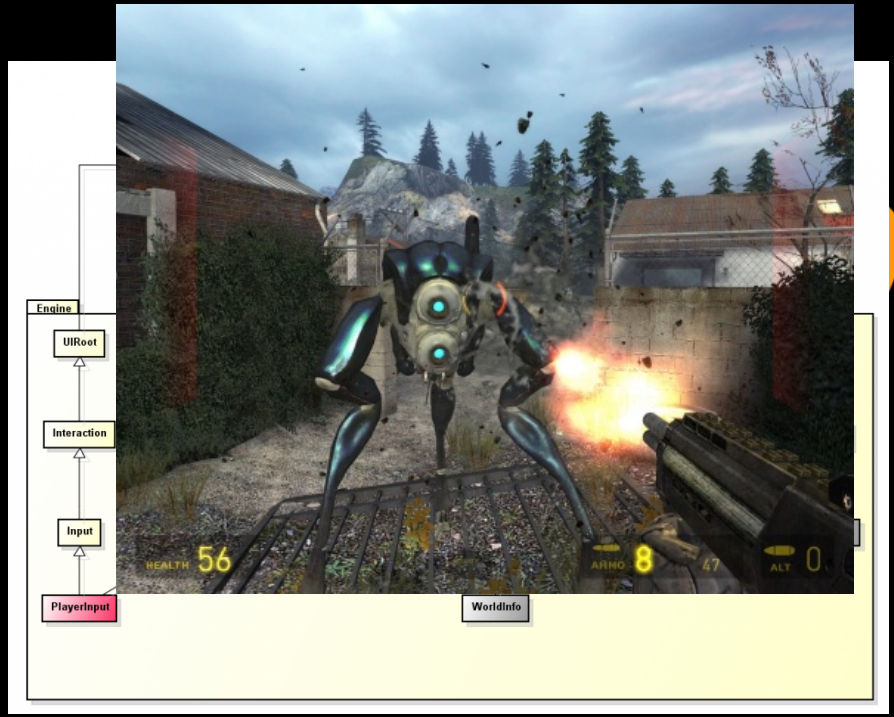
WELCOME TO CS1972!

Introduction

STAFF

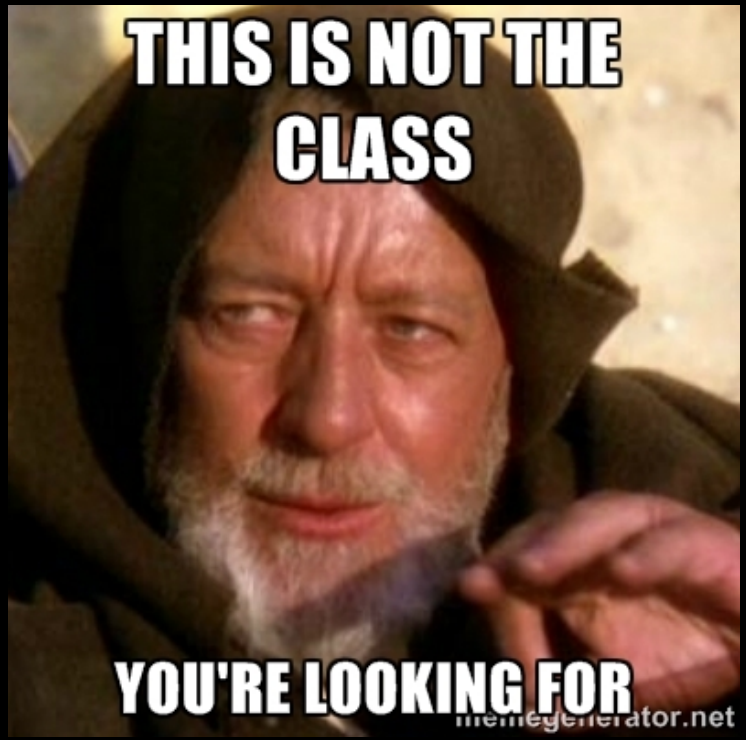
Class Goals

- Build your own 3D game engine, from scratch!
- Build games on top of your game engine!
- Improve your software engineering and design skills!



Not Class Goals

- Using existing 3D game engines such as Unity, Unreal, or XNA (try online tutorials)
- Advanced rendering techniques (try CS224)
- Implementing realistic physics models like rigid bodies, fluid motion, or ragdoll physics (try either of the above)
- Game design (try Brown Gamedev)
  - We recommend the “Extra Credits” YouTube series – the channel is linked on the Docs page!
- Networking models or advanced AI
  - At least not until the final project



What is a game engine?

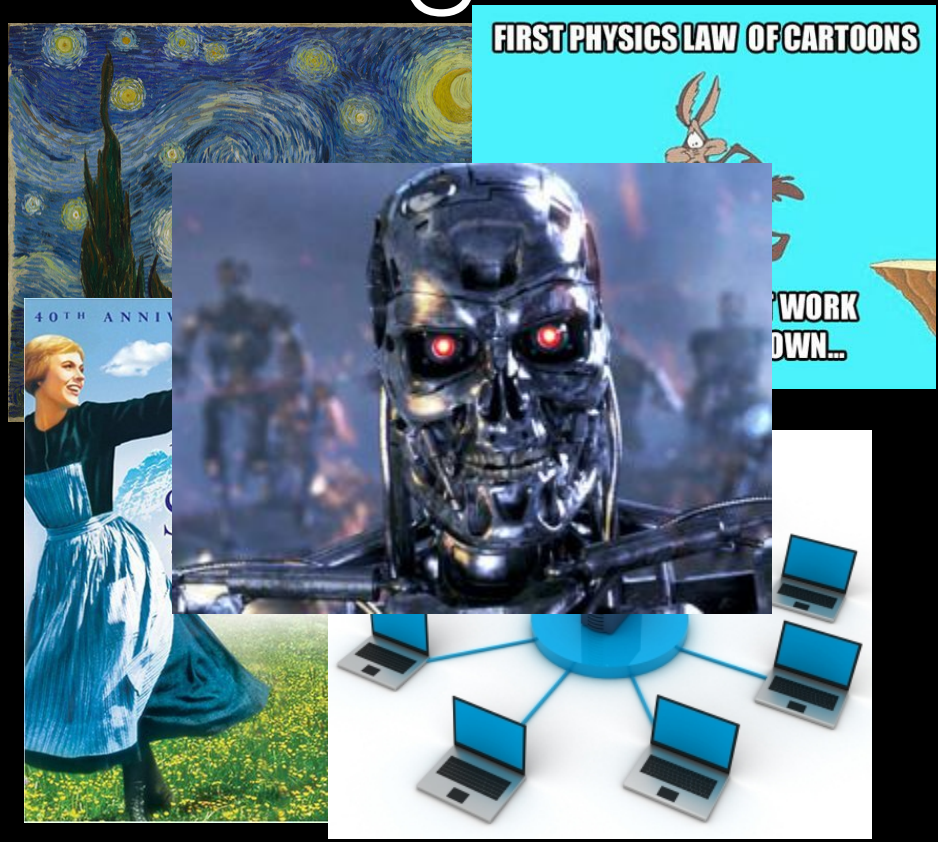
- "The things that games are built on" – zdavis, creator of CS1971/CS1972
- Games have a ton of functionality in common
  - Even beyond superficial things typically defined by genre or gameplay
- Why re-write those same sets of functionality every time we want to make a new game?

What is a game engine?

- Generalizable system providing a set of useful, flexible components
  - Generalizable - “Could other games use this functionality?”
  - Useful - “Would other games benefit from using this functionality?”
  - Flexible – “Can a game tweak how the system behaves to get a desired effect?”
- Implemented as a framework or a library
  - Frameworks dictate control flow
  - Libraries do not dictate control flow
- Your engine will use both design patterns

What is a game engine?

- Systems an engine might support:
  - Rendering
  - Input
  - Physics
  - Sound
  - Networking
  - AI
  - And much, much more
- Each of these serves a dramatically different purpose depending on the game using it
- Can we make an engine that supports every kind of game?



Real-time strategy (RTS) engine

- Large number of low-detail game units
- Multiple levels of AI (unit, squad, team)
- Client/server networking with some lag tolerance
- Heightmap-based terrain

Vehicle simulation engines

- Low number of high-detail models, with level-of-detail management
- Limited AI components
- Minimal network latency
- Realistic environment and physical forces

The universal game engine

- It doesn't exist
- We can't build all of those systems in one semester
  - Most industry engines have been in development for years
- So what will we focus on?

Collisions

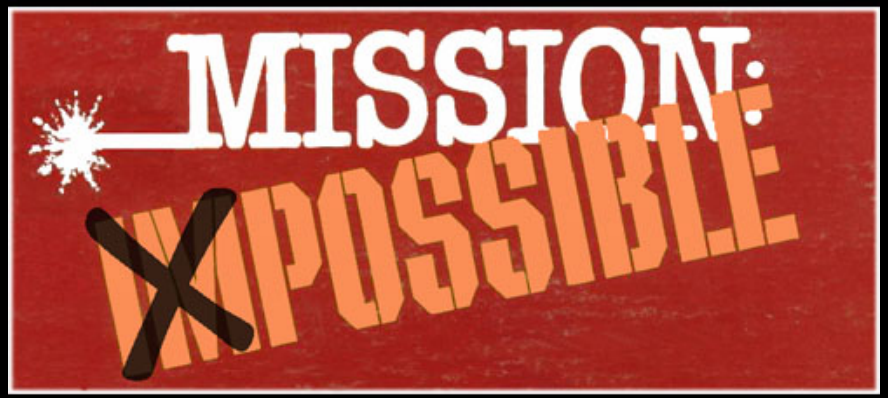
- Most games use collisions in some way
  - 3D games especially
  - Can you think of any that don't?
- We will build:
  - A "common" engine supporting game-agnostic functionality
  - Two collision engines, "voxel" and "geometric", supporting concrete world representations

Introduction

ASSIGNMENTS

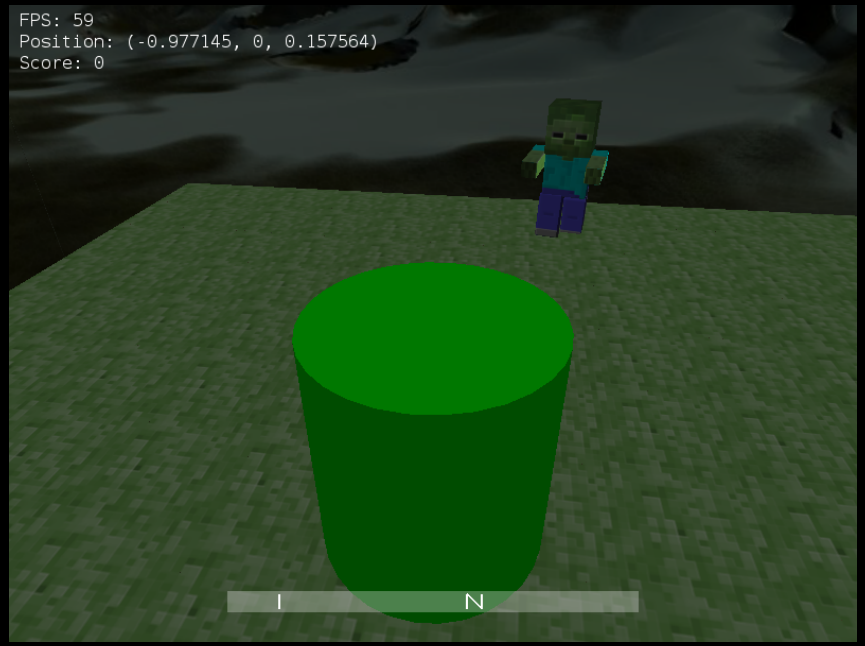
Assignments

- Three solo assignments split up into weekly checkpoints
  - Design and implement engine features and a game to demonstrate functionality
  - Iterate over several weeks to build a final product
- One open-ended final group project
  - Add some advanced functionality and show off your engine



Warmup: 2 Weeks

- Startup assignment to get familiar with working in 3D space
  - Basic engine architecture
  - Game world representation
  - Inter-object collisions
- Limited gameplay, but plenty of room for creativity!

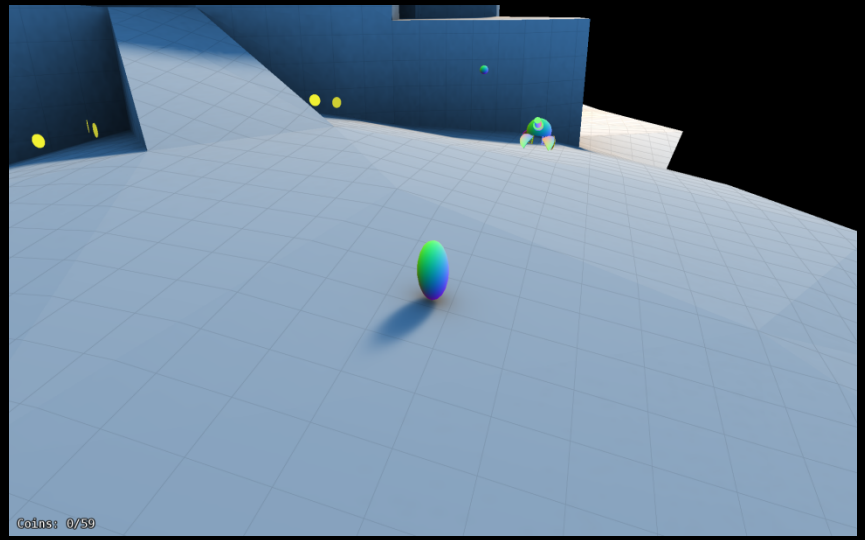


Minecraft: 3 Weeks

- Open world game built on top of your “voxel” engine
  - Block and chunk-based world representation
  - Procedural terrain generation
  - Collisions and raycasting
- Make the equivalent of the real Minecraft Alpha in three weeks!

Platformer: 4 Weeks

- Open-ended game built on your “geometric” engine
  - Externalized polygonal world representation
  - Collisions and raycasting
  - Pathfinding and string pulling
  - Heads up displays
- Take a week to make some additional gameplay!



Final: ~4 Weeks

- Pick advanced engine features and design gameplay
  - Pitch to TA's for a mentor
  - Pitch to the class to find teammates
- Groups strongly encouraged
- Public playtesting required, polish recommended
- All yours afterwards

Introduction

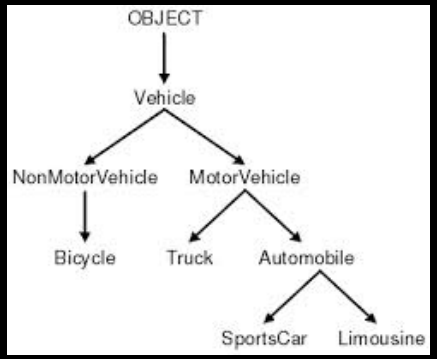
TIMES AND REQUIREMENTS

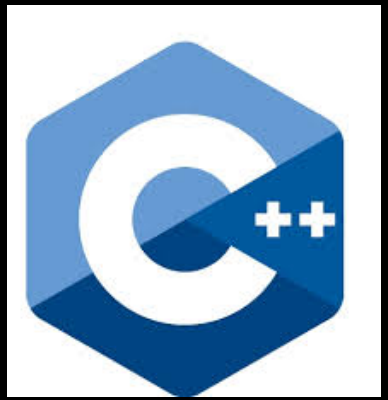
When is class?

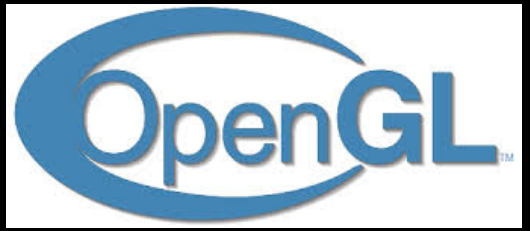
- Lecture 3:00-5:20pm, Wednesday in Lubrano
  - 1-1.5 hours of lecture
  - 20 minutes of “weeklies”
  - 20 minutes of playtesting
- Design Checks: Thursday to Saturday
- TA Hours: Sunday to Tuesday
- Assignments due every Tuesday

To take this class, you must…

- Have taken one of the following courses (in order of usefulness):
  - CS123 or CS195n/CS1971
  - CS32 or CS33
- Be very comfortable with object-oriented design
- Be very comfortable with programming large C++ projects
- Be comfortable with OpenGL, meaning you have:
  - Taken CS123 OR
  - Completed our OpenGL bootcamp







In addition, it helps if you…

- Are comfortable with vector math and geometric relationships/ definitions
- Are familiar with a version control system
- Have played a diverse set of 3D games

Introduction

GRADING

Simple Grading System

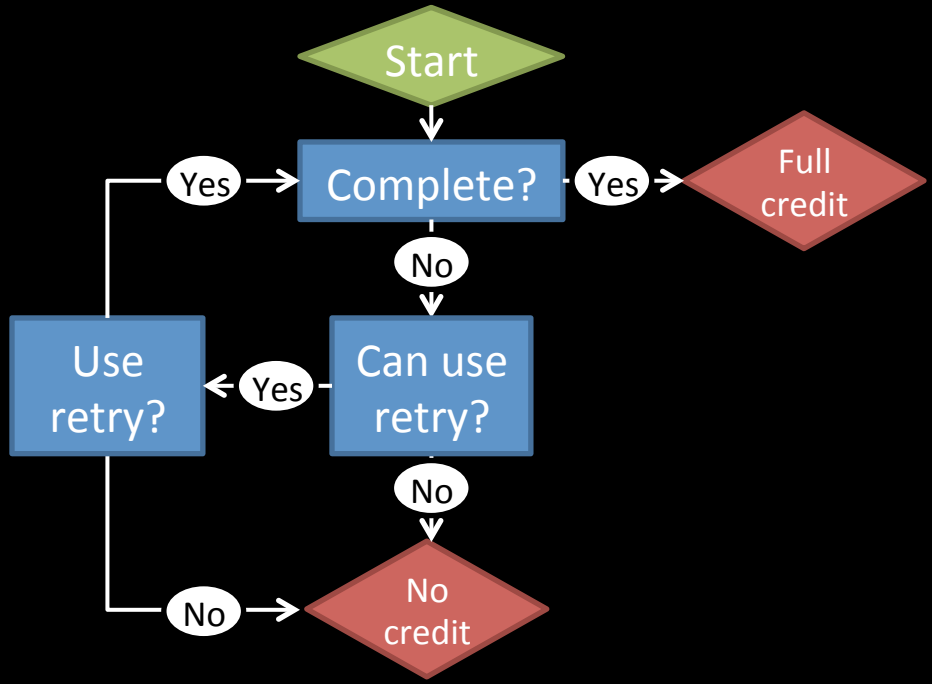
- Four projects, no HW/exams
  - Warmup, Minecraft, Platformer, and Final
  - Broken down into weekly checkpoints
  - Handin due every Tuesday at 11:59:59 PM
- If your handin meets all requirements, it is “complete”, otherwise it is “incomplete”

Requirements include…

- Global requirements
- Weekly checkpoint requirements (split up between “engine” and “game” requirements)
- Design checks
  - Sign up using cs1972_signup <project>
  - High level conceptual questions, but not a free pass
- Playtesting other students’ games
  - Help find bugs, give feedback on gameplay
  - Only for weeks with “real” gameplay

I got an incomplete, now what?

- "Retry" within a week of receiving grade
  - "Standard" retry – one per checkpoint, don't carry over
  - "Extra" retries – two for the entire class, due 1 week from receiving standard retry grade
- Successful retry results in full credit on that checkpoint
  - Otherwise, that checkpoint is a "no credit"
- E-mail the TA's when you hand in a retry!
  - We may grade it earlier
  - If you don't, we may miss your retry



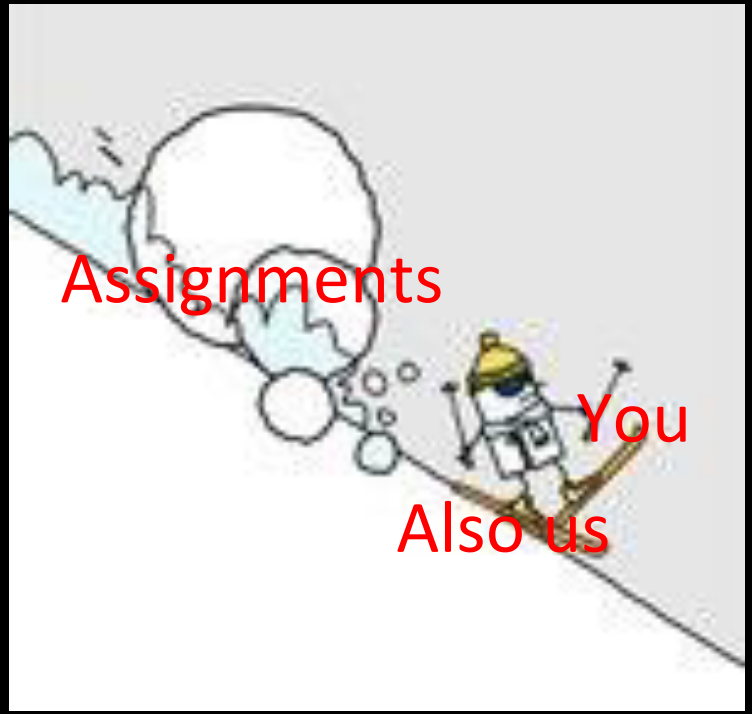
Final Grades

- Final grade determined by number of no-credits at the end of the semester
- There is no curve!
  - Do the work, get an A!
- Must hand in a working version of all checkpoints to pass!

# NC	% complete	Grade
0	100%	A
1	93%	A
2	86%	B
3	79%	C
4	71%	C
5+	<70%	NC

A word of warning…

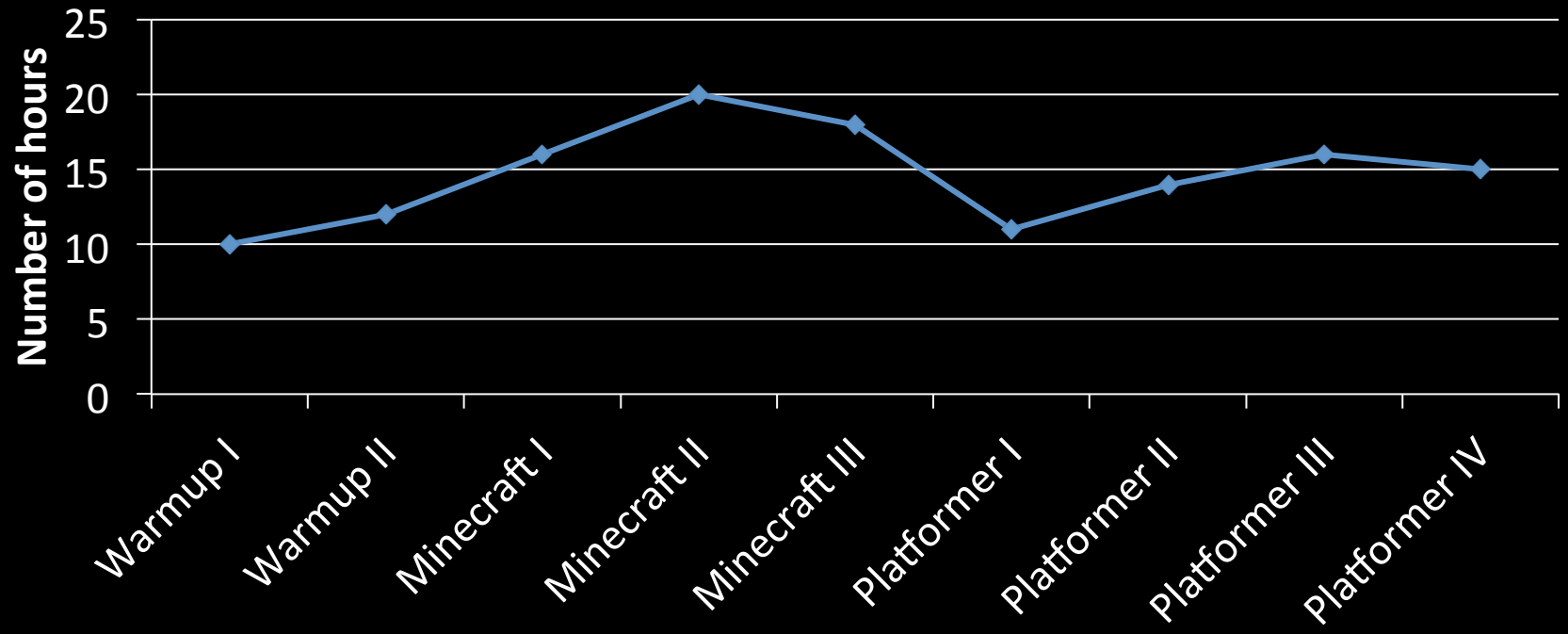
- The retry system is there to help you, but abusing it will cause the “snowball of death”
- Grading late/retry handins puts more stress on the TA’s
- If your handin is playable in some way, turn it in even if you know it isn’t complete
  - You’ll get playtesting feedback
  - You’ll get TA feedback
- If it isn’t, go to sleep!



Introduction

DIFFICULTY

Expected hours



Introduction

ABOUT REGISTRATION

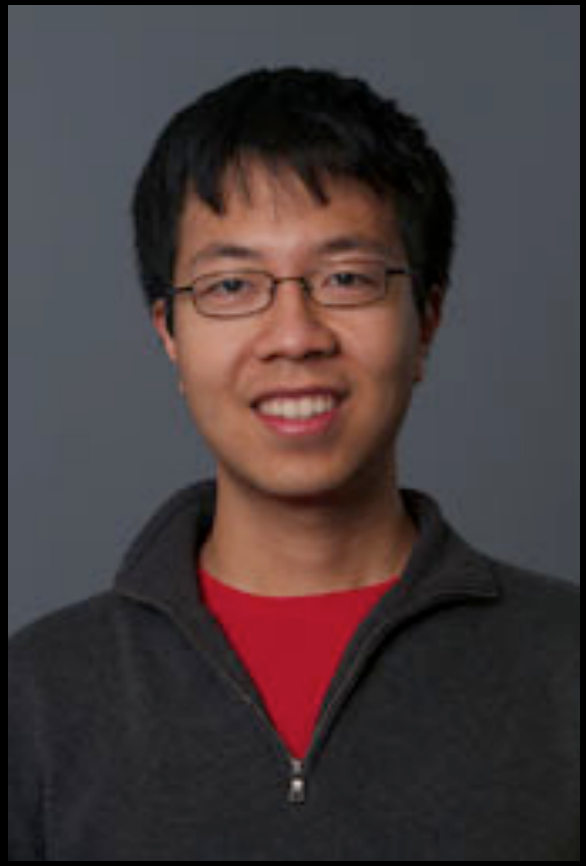
Registering for CS1972

- This course is an independent study, so we have multiple sections
- We will tell you which section to sign up for and get you a registration code next Wednesday
- **If you have any interest in taking this course, please give your name to the TA's on the way out!**
  - You will not be assigned a section if you do not do this

Introduction

QUESTIONS?

A word from our sponsor



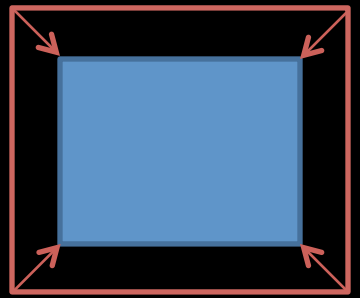

LECTURE 1
Basic Engine Architecture
(Common Engine)

Basic Engine Architecture (Common Engine)

THE MOST BASIC INTERFACE

A game generally needs…

- Timed updates (“ticks”)
- Ability to render to the screen (“draws”)
- Input events (in some form or another)
- Knowledge that it has been resized (not that important for this class)



Ticks

- General contract:
  - `void tick(long nanos)`
  - Nanos is the most precision most computers have
  - Tip: Many people prefer to convert to `float seconds`
  - Your support code already does this
- Simply notifies the engine that a given amount of time has elapsed since the previous “tick”
  - But this is hugely important
  - Nearly all logic takes place during “ticks”
- Updates per second (UPS) is how many ticks occur in a second
  - Keeps track of how smoothly the game world is updated
  - We require 20 UPS in all projects

Draws

- General contract:
  - `void onDraw(Graphics *g);`
  - Convert game state into viewable form
  - Many people make their own `Graphics` object to abstract out common drawing methods
- MUST BE FREE OF SIDE EFFECTS!
  - Two subsequent draw calls should produce identical results

Input Events

- Exact contract differs depending on type, but usually of the form:
  - `void onDDDEEE(QDDDEvent *event);`
  - DDD = device type (e.g. mouse, key)
  - EEE = event type (e.g. moved, pressed)
- Event object contains information about the event
  - e.g. how far the mouse moved; what key was pressed...

Putting it together

- Basic methods of a game application:
  - (note: support code calls these, you implement them)

```
class Application {
public:
        void onTick(long nanos);
        void onDraw(Graphics *g);
        void onKeyPressed(QKeyEvent *event);
        // more device and event types here...
        void onMouseDragged(QKeyEvent *event);
}
```

The Most Basic Interface

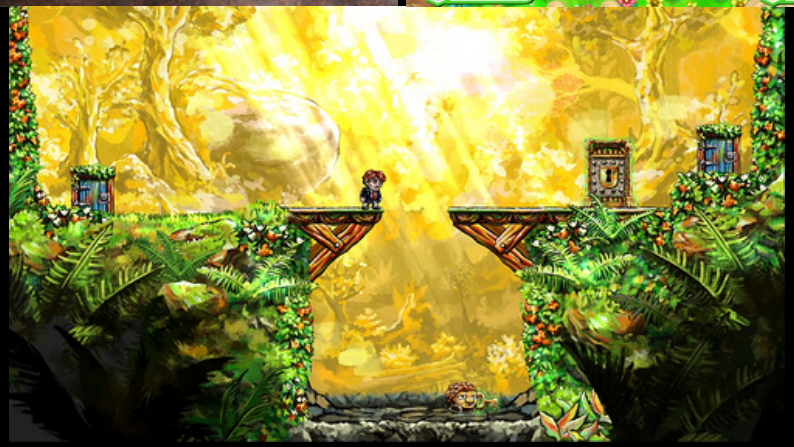
QUESTIONS?

Basic Engine Architecture (Common Engine)

APPLICATION MANAGEMENT

We know the external interface

- But how does one build an engine around that?
- Drawing/ticking/event handling is very different depending on what's going on!
  - Menu system
  - The actual game
  - Minigames within game

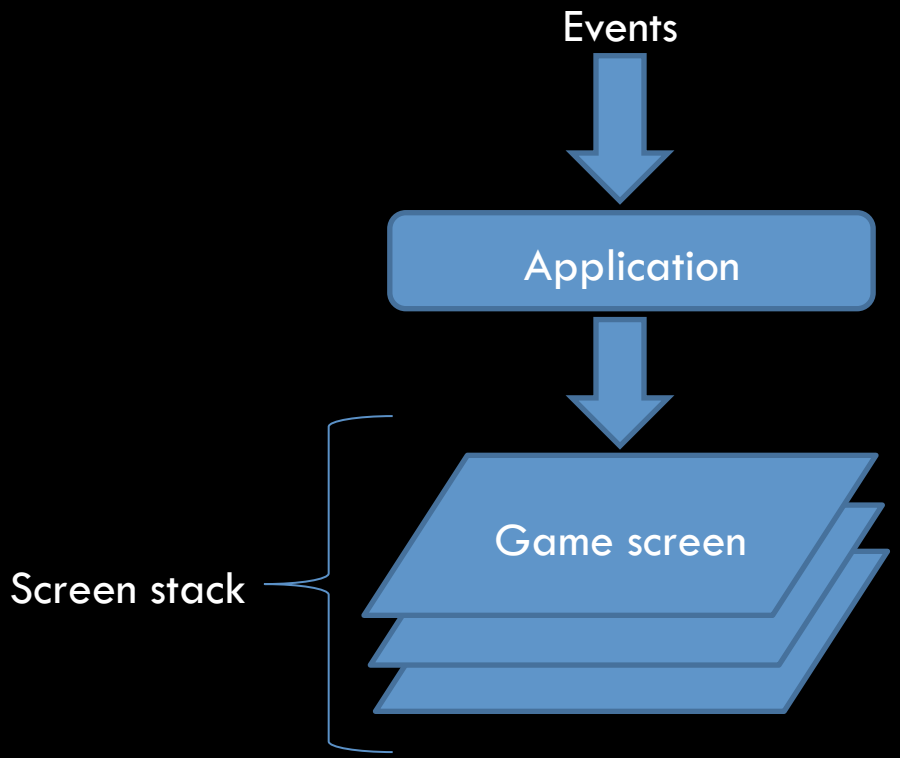


Solution: `Screen`s within `Application`

- Rather than keeping track of “modes”, separate each game screen into a dedicated `Screen` subclass
- A `Screen` has similar methods to the `Application`
  - `onTick`
  - `onDraw`
  - Input event methods

Keeping track of `Screens`

- Simplest way:
  - Single `Screen` in `Application` at a time
  - Current `Screen` calls `setScreen()` on `Application`
- Alternative way:
  - Stack of `Screens` maintained by the `Application`
  - Topmost `Screen` gets events
  - Advanced: "Transparent" `Screens` can forward calls down to other `Screens`



Application Management

QUESTIONS?

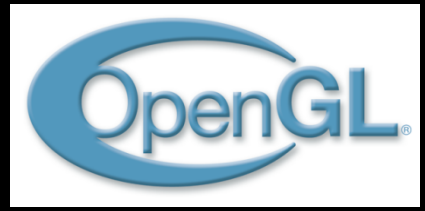

LECTURE 1
Camera and Graphics
(Common Engine)

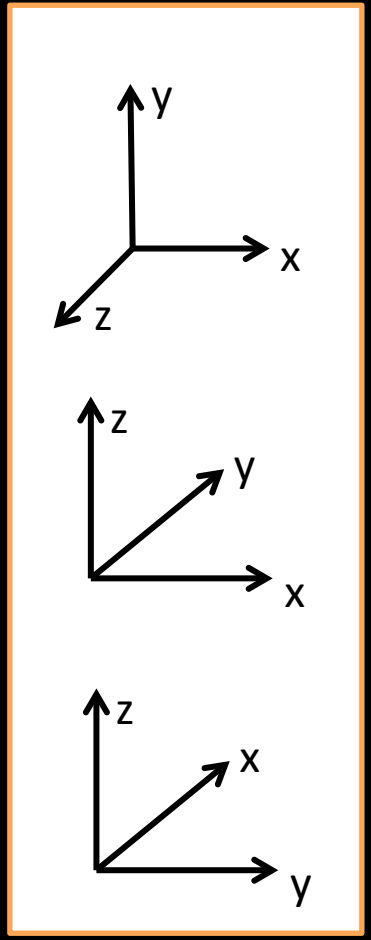
Camera and Graphics (Common Engine)

FIRST PERSON CAMERA

Coordinate systems

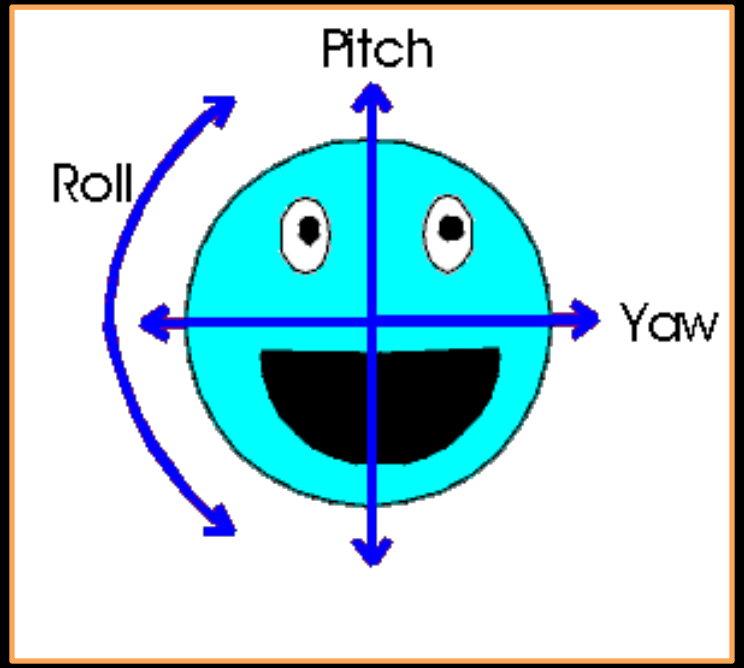
- Different game engines define coordinate systems differently
  - Most of you will probably use the OpenGL coordinate system
- TAs will strive to be coordinate-system independent
- "Horizontal plane"
  - Plane parallel to the ground (in OpenGL, the xz-plane)
- "Up-axis"
  - Axis perpendicular to horizontal plane (in OpenGL, the y-axis)





Yaw/Pitch

- Yaw
  - Stick a pin in the top of the player and rotate them around it
- Pitch
  - The player looking up and looking down
- Roll
  - Only really used in flight simulators
- Easier than u,v,w



Horizontal player movement

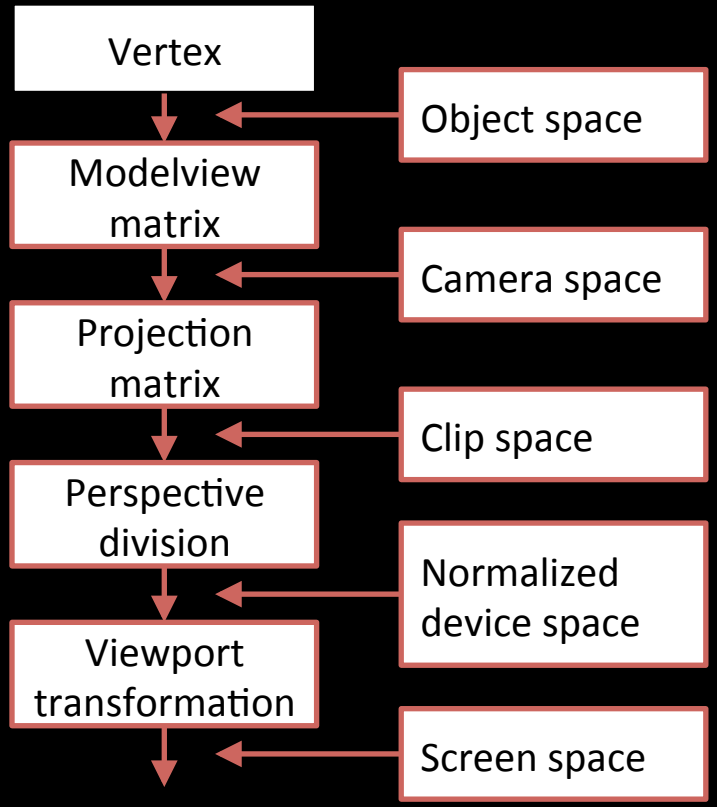
- Simple trigonometry for horizontal movement
  - Player is rotated (yawed) around the up-axis at some angle
  - Get vector of forward velocity in horizontal plane
  - Multiply velocity by time since last tick and add to position
- Strafing
  - Same as above, but use angle 90° left or right from the player's facing direction

Vertical player movement

- Acceleration due to gravity
  - $g=$ (a reasonable negative constant)
  - $p\downarrow up \mathrel{+}= tv\downarrow up + .5gt\uparrow 2$
  - $v\downarrow up \mathrel{+}= gt$
- Collision with ground
  - After previous step: $p\downarrow up = \max(0, p\downarrow up)$
  - $v\downarrow up = 0$

OpenGL matrix transformations

- Not the same as the five matrices from CS123
  - OpenGL conflates them into two matrices (projection, modelview)
  - Stored in column-major order rather than row-major
- Modelview matrix
  - Transforms object space to camera space, usually changes every frame
- Projection matrix
  - Has camera parameters (aspect ratio, field of view), usually only changes on window resize



First-person camera

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(fov, ratio, near, far);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(eye, center, up);
```

- `ratio` is aspect ratio (width/height of screen)
- `fov` should be between 0 and 180
- `gluLookAt()` takes 9 arguments, 3 for each vector

First Person Camera

QUESTIONS?

Camera and Graphics (Common Engine)

BASIC GRAPHICS

Motivation

- Certain drawing calls are common to many games
  - Loading a texture
  - Drawing a texture
  - Drawing a primitive shape
- We can store all of our texture ids in one centralized object
  - Helps us not load them into memory more than once
  - Helps us keep track of them and delete them
- We can encapsulate these in a “Graphics” object

Texture loading

- Easiest to leverage Qt framework:
  - Remember to delete textures!

```
QImage img(path);
img = img.mirrored(false, true);
unsigned int id;
id = view.bindTexture(img, GL_TEXTURE_2D, GL_RGBA,
                QGLContext::MipMapBindOption |
                QGLContext::LinearFilteringBindOption);
```

Texture loading

- If you'd rather use straight GL calls:

```
QImage img(path);
img = QGLWidget::convertToGLFormat(img);

unsigned int id;
glGenTextures(1, &id);
glBindTexture(GL_TEXTURE_2D, id);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, img.width(), img.height(),
        0, GL_RGBA, GL_UNSIGNED_BYTE, img.bits());
```

Drawing geometry

- In this class, we will usually use immediate mode:
  - If you are more familiar with the newer OpenGL API's, feel free to use those
  - However, the staff is only familiar with the old API

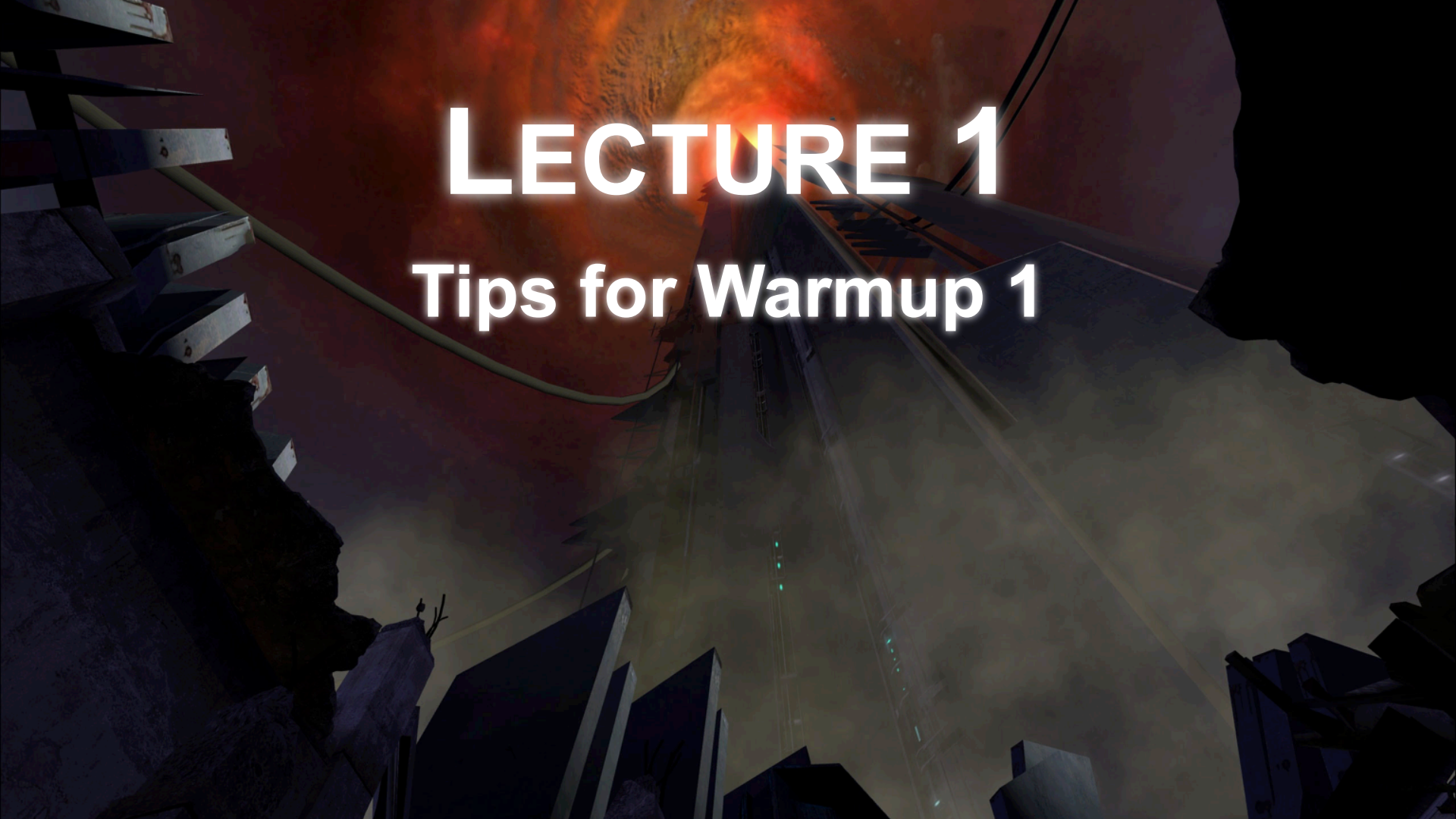
```
glBindTexture(GL_TEXTURE_2D, id);
glEnable(GL_TEXTURE_2D);
glBegin(GL_QUADS);
glTexCoord2f(0,0); glVertex3f(0,0,0); // remember winding order!
glTexCoord2f(1,0); glVertex3f(1,0,0);
glTexCoord2f(1,1); glVertex3f(1,1,0);
glTexCoord2f(0,1); glVertex3f(0,1,0);
glEnd();
glDisable(GL_TEXTURE_2D);
```

Graphics Object

- For this week, we require you to do texture loading, storing, drawing, and deleting in your graphics object
- Use it for any method you think you might want to use somewhere else in your code or in another game

Basic Graphics

QUESTIONS?


LECTURE 1
Tips for Warmup 1

Support Code Overview

- Qt Framework
  - main.cpp – starts up program, toggles fullscreen
  - mainwindow.h/.ui/.cpp – sets up window
  - view.h/.cpp – basic even framework, where your Application class should reside
- Vector math – vector.h
  - 2,3,4 dimensional vectors
  - Tons of math
  - Good place to put convenience functions (like frand(), max(), min(), etc…)
- QRC files
  - Allows for easy access of external resources
  - Use for texture loading this week

```
12  public:
13      View(QWidget *parent);
14      ~View();
15
16  private:
17      QTime time;
18      QTimer timer;
19
20      void initializeGL();
21      void paintGL();
22      void resizeGL(int w, int h);
23
24      void mousePressEvent(QMouseEvent *event);
25      void mouseMoveEvent(QMouseEvent *event);
26      void mouseReleaseEvent(QMouseEvent *event);
27
28      void keyPressEvent(QKeyEvent *event);
29      void keyReleaseEvent(QKeyEvent *event);
30
31  private slots:
32      void tick();
```

Support Code Overview

- Methods in view.h/.cpp
  - onDDDEEE(QEEEvent *event) – call app.dDDEEE(event)
  - onTick(float seconds) – call app.tick(seconds)
  - onDraw() – call app.draw()
  - initializeGL() – can't make OGL calls until here
  - onResize(int x, int y) – call glViewport(), inform camera
- Make Application a separate class from View!

Drawing Shapes

ansform

- ant
- gon
- Fe x each
- paste code a ce
  ices every time

With Transforms

- Write a method to draw a standard 1x1 quad in the horizontal plane
- Determine the transforms needed for this particular one
- ???
- Profit!
- Recalculate transforms for each new shape


LECTURE 1
C++ Tip of the Week

Qt vs. STDLib

- QString – substrings, splitting, hashcodes
- QList – type-generic dynamic array
- QHash – type-generic hashtable
- QSet – type-generic set
- QTimer – sets up the game loop
- QThread – easy-to-use threading API
- QPair – great for Vector hashcodes

http://qt-project.org/doc/qt-4.8/qtcore.html

Initializing instance members

```
class MyClass {
public:
  // Usually prints garbage
  MyClass() { if (check) cout << x + y << endl; }
  // Always prints garbage
  MyClass() {
    check = 1; x = 3;
    if (check) cout << x + y << endl;
  }
  // Always prints 8
  MyClass() {
    check = 1; x = 3; y = 5;
    if (check) cout << x + y << endl;
  }
private:
  bool check; int x, y;
};
```

- Remember to initialize any and all instance variables!
  - Bugs will make no sense
  - Things will fly off the screen
  - You will get frustrated


LECTURE 1
C++ Anti-Tip of the Week

Backwards Iteration is Best!

```
int *arr = new int[100];
for (int i = 0; i < 100; i++) {
    arr[i] = i;
}
// Prints 42
cout << 42[arr] << endl;
```


WARMUP1 IS OUT!
Sign up for design checks!
Good luck!