LECTURE 1

Introduction
WELCOME TO CS1950U!

Introduction
Introduction

STAFF
Introduction

GOALS AND REQUIREMENTS
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!
To take this class, you should...

• Be comfortable with object-oriented design
• Be comfortable with programming large projects
• It is useful to have some experience with 3D graphics and C++, but not required
• It is useful if comfortable with 3D geometry and some basic vector math, but not required
ASSIGNMENTS
Projects

- Three projects split up into weekly checkpoints
  - One checkpoint every week
- One open-ended final group project
Warmup

• Startup assignment to get familiar with working in 3D space
• 2 week project
• Basic UI, Player Controls, Collisions, Enemies
Minecraft

• Basically alpha release of Minecraft!
• 3 week project
• Spatial partitioning, procedural terrain generation, collisions and raycasting
Platformer

- Open-ended game
- 4 week project
- Rigid body physics, pathfinding, heads up displays, UI
Final

- ~5 week project
- Your choice of engine features
- Your choice of game features
- Groups recommended
- Public playtesting
- More details later
Introduction

GRADING
Simple Grading System

• Only projects
• Every project broken down into weekly checkpoints
  – Handin due every Tuesday at 11:59:59 PM
• Each checkpoint worth 4 points
Simple Grading System

• For each project, you have…
• Playtesting requirements (1 pt)
  – Constitute a basic, playable demo
• Primary requirements (2 pts)
  – Major features
• Secondary requirements (1 pt)
  – Less important features
• Global requirements (0 pts)
  – Required for every project
Final Grades

• No curve!
  – Do the work, get an A

• Still need to meet all primary engine requirements to pass the class

<table>
<thead>
<tr>
<th>Points</th>
<th>Missing</th>
<th>Percent</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>46-54</td>
<td>0-8</td>
<td>85</td>
<td>A</td>
</tr>
<tr>
<td>35-45</td>
<td>9-19</td>
<td>65</td>
<td>B</td>
</tr>
<tr>
<td>&lt;34</td>
<td>&gt;20</td>
<td>&lt;63</td>
<td>C</td>
</tr>
</tbody>
</table>
Design Checks

- Sign up with cs195u_signup <project>
- High-level conceptual questions
- Gives one standard retry, which bring us to ...
Incomplete Handins

• Standard Retry
  – As long as you complete a design check, you are allowed to re-hand in a checkpoint

• Extra retries
  – You have two for the whole class
  – Can use to retry a checkpoint that you already retried

• You have a week to use each retry
Incomplete Handins

• Playtesting requirements cannot be retried
• No credit for secondary requirements until primary requirements are met
• Only your best handin will count (retries never hurt your grade)
Out of Retries

• Used the standard retry, out of extra retries, now what?
• You can still do well in the class
  – Don’t have to get credit for all requirements
• You can still pass the class
  – Hand in working version of all primary requirements by the end of the semester
Please hand in on time!

• Falling behind causes a “snowball of death”
• Grading late handins puts stress on the TAs
• If your handin is playable, hand it in even if you’re missing some reqs so you can be playtested
• If it isn’t, go to sleep! You have another week to retry
Introduction

CLASS TIMES
Class Times

• Class: Wednesday 3:00pm-5:00pm
  – Lecture 1-1.5 hours (CIT 506)
  – Playtesting ~30 minutes (Sunlab)

• Design Checks: Thursday-Saturday
  – Sign up using cs195u_signup <project>

• TA hours: Sunday-Tuesday

• Website: http://cs.brown.edu/courses/cs195u/
ABOUT REGISTRATION
Registering for CS1950U

• If you are not yet registered, please give your name to the TA’s on the way out!
QUESTIONS?
A word from our sponsor
LECTURE 1
Basic Engine Architecture
WHAT IS A GAME ENGINE?

Basic Engine Architecture
What is a game engine?

• “The things that games are built on” – Jeffrey Hao
• Games tend to have a ton of functionality in common
• Solution: create “engines” that abstract out common functionality
What is a game engine?

• Should be usable by many games
  – If you gave your engine to someone, they should be able to easily create a game without modifying engine code
• Should be general
  – No game-specific logic!
What does this look like?

• Sample package hierarchy
  – src/
    • engine/
      – Screen.cpp
      – Screen.h
    • warmup/
      – WarmupScreen.cpp
      – WarmupScreen.h
Basic Engine Architecture

AN ESSENTIAL INTERFACE
A game generally needs...

- Timed updates ("ticks")
- Ability to render to the screen ("draws")
- Input events (in some form or another)
- Knowledge that screen has been resized
Ticks

- General contract:
  - `void tick(float seconds)`

- Tells the game that a given amount of time has elapsed since the previous “tick”
  - This is hugely important
  - Nearly all logic takes place during “ticks”
Draws

• General contract:
  – void draw(Graphics *g);

• Tells the game to draw itself
  – Convert game state into viewable form
  – No game logic should take place in draw calls

• More information coming up in Graphics section
Input Events

• Most APIs provide input events rather than making you manually poll mouse and keyboard

• 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)

• Tells the game that an event has occurred
  – Event object contains information about the event
    • e.g. how far the mouse moved; what key was pressed...
Putting it Together

• The **Application** class

```cpp
class Application {
public:
    void tick(long nanos);
    void draw(Graphics *g);
    void onKeyPressed(QKeyEvent *event);
    // more device and event types here...
    void onMouseDragged(QKeyEvent *event);
}
```
Putting it Together

• Application represents an instance of a game
• You will implement Application in Warmup1
The Most Basic Interface

QUESTIONS?
Basic Engine Architecture

SCREEN MANAGEMENT
We have an Application

- But how do we build a game around that?
- Drawing/ticking/event handling is very different depending on what’s going on!
  - Menu system
  - The actual game
  - Minigames within game
Solution: Screens within Application

• Rather than keeping track of “modes”, separate each “mode” into a dedicated Screen subclass
  – MenuScreen, GameScreen, etc.

• A Screen has similar methods to the Application
  – tick
  – draw
  – Input event methods
Keeping track of **Screens**

- **Simplest way:**
  - Single *Screen* in *Application* at a time
  - *Application* forwards all events to this screen

- **Alternative way:**
  - Map of *Screens* maintained by the *Application*
  - One active *Screen* gets events (most of the time) or multiple screens can get these events (special case)
What are Screens good for?

• For Warmup1, Screens should
  – Draw the entire game
  – Handle all of the game logic
• In general, Screens shouldn’t do this
  – Results in serious spaghetti code
• Solution: GameWorld
  – Covered next week...
QUESTIONS?
LECTURE 1
Camera and Graphics
Camera and Graphics

CAMERA
Cameras

- Physical camera will render a “film” — a 2D representation of the 3D space
- For virtual cameras, goal is similar
  - Render by squashing view volume (or frustum) onto 2D plane
Cameras in 3D Space

• Camera is not very useful unless we know
  – Where it is (position)
  – What it’s orientation is (pitch, yaw)
Camera Orientation

- **Yaw**
  - Stick a pin in the top of the camera and rotate it around it by this angle

- **Pitch**
  - The camera looking up and looking down by this angle

- **Roll**
  - Only really used in flight simulators
Camera Orientation

• Alternatively...
• Specify direction the camera is facing as a vector
  – Called the “look vector”
Camera position

- Position of camera in the world
- For Warmup 1, in order to achieve first person...
  - Make camera position same as player position
  - Update camera position to make the same as player position
Other Camera Parameters

• Field of view angle
  – How wide is the view volume?

• Aspect ratio
  – Ratio of the width of screen to the height of the screen
Our Camera Class

• Default Camera class provided
  – src/engine/graphics/Camera.h/cpp

• Allows you to specify all of the above attributes
  – Most likely will only modify position, pitch, yaw
First Person Camera

QUESTIONS?
Motivation

- Certain graphics calls are common to many games
  - Setting up a camera
  - Drawing shapes
  - Setting material properties for shapes
  - Drawing text
- We can store all of our shapes, materials, fonts, etc. in one centralized object
  - Helps us not load them into memory more than once
  - Helps us keep track of them and delete them
- Encapsulated in a “Graphics” object
Graphics Object

• Default Graphics object provided
  – src/engine/graphics/Graphics.h(cpp)

• Methods for …
  – Setting the active camera
    • This camera will be used for rendering
  – Drawing shapes
    • Rectangles (quads), cylinders, and spheres for now
  – Setting materials
    • Change color, texture, lighting of shapes
  – More!
Other Classes

• src/graphics/Shape.h(cpp)
  – Describes the geometry of a shape
• src/graphics/Material.h(cpp)
  – Describes material properties of a shape
• More!
Doing it Yourself

• Feel free to modify graphics support code!

Support Code B200

An error has occurred. The printer needs repair. Follow the steps below.
1. Turn the device off and unplug the power cord from the power supply.
2. See the manual and contact the service center.
QUESTIONS?
LECTURE 1
Controls
PLAYER MOVEMENT

Controls
Coordinate systems

• Different game engines define 3D coordinate systems differently

• In your case...

• “Horizontal plane”
  – Plane parallel to the ground (the xz-plane)

• “Up-axis”
  – Axis perpendicular to horizontal plane (the y-axis)
Horizontal player movement

- Keep track of player position $pos$
- Forward movement
  - Use the horizontal component ($xz$) of the camera look vector
  - $dir = \text{normalize}(\text{look}.x, 0, \text{look}.z)$
  - $pos += dir$
- Strafing
  - Same as above, but use angle 90 degrees left or right from the camera’s look direction
  - $perp = (\text{dir}.z, 0, -\text{dir}.x)$
  - $pos += perp$
Vertical player movement

• Keep track of player vertical position $pos_{up}$
• Keep track of player vertical velocity $velocity_{up}$
• Apply acceleration due to gravity
  – $dt = (\text{time since last tick})$
  – $g = (\text{a reasonable negative constant})$
  – $velocity_{up} += g \cdot dt$
  – $pos_{up} += velocity_{up} \cdot dt$
• Collision with ground
  – After previous step: $pos_{up} = \max(0, pos_{up})$
  – $velocity_{up} = 0$
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 — glm (important!)**
  - 2, 3, 4 dimensional vectors and matrices
  - Tons of math — see online documentation

• **QRC files**
  - Allows for easy access of external resources
  - Can use to load your own resources
Support Code Overview

• Utility
  – src/engine/util/CommonIncludes.h
    • Includes glm, iostream
    • Include this anywhere you need glm
• Graphics
  – src/engine/graphics/*
  – Described in previous section

```cpp
#define COMMONINCLUDES_H
#include "GL/glew.h"
#include <iostream>
#define GLM_FORCE_RADIANS
#include <glm/glm.h>
#include <glm/gtx/string_cast.hpp>
#include <glm/gtx/transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtc/constants.hpp>
#endif // COMMONINCLUDES_H
```
Support Code Overview

- **Methods in view.h/.cpp**
  - DDDEEEEvent(QEEEEvent *event) – call app.DDDEEE(event)
  - tick(float seconds) – call app.tick(seconds)
  - paintGL() – call app.draw(graphics)
  - resizeGL(int x, int y) – call app.resize(dimensions)

- **Make **Application** a separate class from View!**
  - Put instance of **Application** class in View, so that you can pass events on to **Application**
Setup Guide

• If you have time, go through the CS1950U setup guide! (highly recommended)
  – On the Docs page of the website
  – We will run through it after class today in the Sunlab

• It covers …
  – How to set up a camera
  – How to draw something using the graphics object
  – How to add basic player controls
On Your Own

• Play around with graphics object calls
• Specifically try to move, resize and rotate shapes
• 3D graphics can be tricky, especially if you haven’t done it before
  – Feel free to email us or come to hours if there’s something you don’t understand
LECTURE 1

C++ Tip of the Week
Qt vs. STLlib

- 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/qtccore.html
Qt vs. C++ STDLib

• QString – std::string
• QList – std::vector
• QHash – std::unordered_map
• QSet – std::unordered_set
• QPair – std::pair

http://qt-project.org/doc/qt-4.8/qtc当地e.html
SMART POINTERS
Raw pointers

• Problems:
  – Declaration doesn’t indicate who owns the object (i.e. who destroys it)
  – Must destroy exactly once
  – Memory leaks
Smart Pointers

• The solution to all of the problems (and more)
  – Most importantly, delete / free object they refer to automatically if pointer goes out of scope

• 3 types in modern C++ (c++14)
  – std::unique_ptr
  – std::shared_ptr
  – std::weak_ptr
Shared Pointers

• In general the one to use
• Same size as raw pointers and perform the exact same instructions
Creating a Shared Pointer

• Use “std::make_shared<T>(args);”
• More verbose than creating a normal pointer, but worth it
Creating a Shared Pointer

• With shared pointers

```cpp
#include <memory>  // Include header file
...
...
std::shared_ptr<Camera> cam = std::make_shared<Camera>();
...
...
```

• Without shared pointers

```cpp
...
...
Camera *cam = new Camera();
...
...
delete cam;
```
Copying a Shared Pointer

• Can make as many copies of a shared pointer as you want
  – `std::make_shared<T> s1 = ...;
  – `std::make_shared<T> s2 = s1;
  – `std::make_shared<T> s3 = s2;
  – ...
  – Each refer to the same object

• Object managed by all shared pointers only deleted when all shared pointers go out of scope
In Summary…

• Basically, you never have to worry about freeing objects again if you use shared pointers
LECTURE 1

C++ Anti-Tip of the Week
Fun with Ternary Operators

```cpp
bool check = frand() > .5f;
int a = (check ? 3 : 4);
int b = (check ? -1 : 1) * 5;
(a > b ? a : b) = 0;
(a < b ? cout : cerr) << (a + b) << endl;
```
WARMUP 1 is Out

Give us your names / logins!
Come to the Sunlab!
Sign up for design checks!
Good luck!