WELCOME TO CS1950U!
Introduction

STAFF
Introduction

GOALS
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!
Useful Skills

- C++
- Graphics/OpenGL
- Basic vector math
Projects

• Two projects split up into checkpoints
  – Some weeks give you choices!

• One open-ended final project (individual or in groups)
Warmup

• Startup assignment to get familiar with working in 3D space
• 2 week project (2 checkpoints)
• Basic engine architecture, graphics, controls
Platformer

• 4 checkpoints over 6 weeks

• Topics:
  – collisions, rigid body physics
  – spatial acceleration
  – pathfinding, AI
  – UI/HUD
  – animation
Final

• 4 week project
• Your choice of engine features
• Your choice of game features
• Groups encouraged, but not required
• More details later
Class Roadmap

Week 1 (Basic Engine Architecture)

Week 2 (Gameworld, ECS, Systems)

Week 3-4 (Ellipsoid/Triangle, Sphere/Cylinder/AAB Collisions)

Week 3-4 (GJK, EPA Collisions)
+ Rigid Body Physics (if you want)
Class Roadmap

Week 5 (Engine Optimizations – spatial subdivision, frustum culling, chunk streaming, texture atlases)

Week 6 (Pathfinding, AI)

Week 7-8 (UI)  Week 7-8 (Skeletal Animation)

Week 9-12 (Final Project)
Introduction

GRADING
Grading

- Only projects
- Grades and feedback will be given on Canvas
- Handins due on Monday at 11:59 PM, except for final, which is due on Sunday 4/18 at 11:59 PM
- Checkpoints are worth 3 points (except for collisions checkpoint which is worth 6 points), final is worth 9 points
Grading

- For each checkpoint, you have...
- Engine requirements
- Game Requirements
- You can get extra credit by implementing extra features
Final Grades

• No curve!
  – Do the work, get an A

• 30 points possible across all projects, not counting extra credit

• Need to complete all primary engine requirements and a final project
## Grading

<table>
<thead>
<tr>
<th>Points</th>
<th>Missing</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>27+</td>
<td>0-3</td>
<td>A</td>
</tr>
<tr>
<td>24-26</td>
<td>4-6</td>
<td>B</td>
</tr>
<tr>
<td>23-</td>
<td>7+</td>
<td>C</td>
</tr>
</tbody>
</table>
Design Checks

- 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 (from when you get your grade back)
Incomplete Handins

• Minimum requirements cannot be retried
• Extra credit can be retried
• No extra credit until all 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 engine requirements by the end of the semester
QUESTIONS?
Introduction

CLASS TIMES
Class Times

• Class: Tuesday 9am-10:20pm (Zoom)

• Design Checks and Hours: Thursday 9am – 10:20pm (Zoom)
  – Optional
  – Signmeup for design checks and hours
  – more hours TBA

• Website: http://cs.brown.edu/courses/cs195u/
Introduction

OTHER COURSE POLICIES
Collaboration Policy

• Full version is on our website

• Short version:
  – Can discuss lectures and assignments
  – Can play each other’s games
  – Cannot look at or give any code
  – Can cooperate with other students during TA hours (at TA discretion)
CS1950U as a Capstone

• Requirements
  – More final project engine features
    • Students taking the capstone should get their project proposals approved before March 22 so that they can start early
    • See the final project handout for details
  – Capstone form filled out, signed by Daniel Ritchie
  – That’s it!
Slack

• We are using Slack instead of Piazza this semester
• Email course staff if you have not been invited to the Slack workspace
• There is a public “help” channel
• You can DM me for private questions
  – I’ll paste questions and answers into the help channel if I think they would be helpful to others (question asker will remain anonymous)
Style Guide

• We expect you to have a reasonable style, but don’t require any specific style guide

• If you’re unsure of what counts as reasonable style, pick your favorite style guide from a course you’ve taken and follow it
Test Your Code

• Your code needs to **compile** and **run** on department machines
  
  – Let me know if there is a problem with FastX

• We can’t grade it if we can’t run it

• Should run at 20+ FPS
Introduction

ABOUT REGISTRATION
Registering for CS1950U

• If you can’t register for CS1950U because you don’t meet the prerequisites
  – Don’t panic
  – Request an override in Courses@Brown
Registering for CS1950U

• If you can’t register for CS1950U because you’re a RISD student
  – Don’t panic
  – Email our professor (Daniel Ritchie)
QUESTIONS?
Introductions!

• Please share
  – Your name
  – Your pronouns
  – A video game you enjoy!
Basic Engine Architecture

WHAT IS A GAME ENGINE?
What is a game engine?

• The things that games are built on
• Games tend to have a ton of functionality in common
• Create engines that abstract out common functionality
What is a game engine?

• Usable by many games
  – It 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 hierarchy
  - src/
    - engine/
      - Screen.cpp
      - Screen.h
    - warmup/
      - WarmupScreen.cpp
      - WarmupScreen.h
What does this look like?

- Engine code should never `#include` game files
Basic Engine Architecture

AN ESSENTIAL INTERFACE
A game generally needs:

- Timed updates (ticks)
- To render to the screen (draws)
- Input events
- Resize events
Ticks

• General contract:
  – `void tick(float seconds)`

• Tells the game that a given amount of time has elapsed since the previous tick
  – Nearly all logic takes place during ticks
  – No drawing should take place during ticks
Draws

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

• Tells the game to draw itself
  – Convert game state into viewable form
  – No side effects from 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(float seconds);
        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 an Application class in Warmup1
SCREEN MANAGEMENT

Basic Engine Architecture
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
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

• **Alternatively:**
  – Map of Screens maintained by the Application
  – Screens can consume events or pass them to a different screen
What are Screens good for?

- For Warmup1, Screens may
  - 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...
Application Management

QUESTIONS?
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 its orientation is (pitch, roll, 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?
Camera and Graphics

BASIC GRAPHICS
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!
- Feel free to write your own graphics code!
Basic Graphics

QUESTIONS?
Coordinate systems

• Different game engines define 3D coordinate systems differently
• Most commonly:
  • “Horizontal plane”
    – Plane parallel to the ground (the xz-plane)
  • “Up-axis”
    – Axis perpendicular to horizontal plane (the y-axis)
Horizontal Movement

• Keep track of your player position

• Forward movement:
  – Use the horizontal component of the look vector
  – forward_speed = some positive constant
  – dir = normalize(look.x, 0, look.y)
  – pos = pos + forward_speed * dir

• Strafing
  – Use the perpendicular of the horizontal direction
  – sideways_speed = some positive constant
  – perp = normalize(dir.z, 0, -dir.x)
  – pos = pos + sideways_speed * perp
Vertical Movement

• Keep track of the player’s vertical position and velocity
• Jump
  – Assign some positive velocity when the player jumps
  – Make sure the player is on the ground (pos.y == 0) before jumping
• Apply gravitational acceleration each tick
  – $dt = \text{time since last tick}$
  – $g = \text{some negative constant}$
  – $\text{velocity} = \text{velocity} + g \times dt$
  – $\text{pos.y} = \text{pos.y} + \text{velocity}$
• Collision with ground
  – After moving the player, set $\text{pos.y} = \max(\text{pos.y}, 0)$
CS195U SUPPORT CODE
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
#ifndef COMMONINCLUDES_H
#define COMMONINCLUDES_H

/*A file for any includes or structs
#include "GL/glew.h"
#include <iostream>
#define GLM_FORCE_RADIANS
#include <glm/glm.hpp>
#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
  – DDDEEEEEvent(QEEEvent *event) – call app.DDDEEE(event)
  – tick(float seconds) – call app.tick(seconds)
  – paintGL() – call app.draw(graphics) or app.draw()
  – 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

• 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
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
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/qtcORE.html
C++ Tip of the Week

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++
  – 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>();
...
...
Camera *cam = new Camera();
...
...
delete cam;
```
Copying a Shared Pointer

• Can make as many copies of a shared pointer as you want
  – \texttt{std::make\_shared<T> s1 = ...;}
  – \texttt{std::make\_shared<T> s2 = s1;}
  – \texttt{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
Avoid Shared Pointer Cycles

• A shared pointer counts how many other objects reference it (i.e. how many copies of the shared pointer exist)
  – When this counter reaches 0, the shared pointer’s destructor is called
• Do not create “cycles” of shared pointers!
  – If a shared pointer s1 owns a shared pointer s2 and s2 also owns a shared pointer of s1, you will get a memory leak!
Avoid Shared Pointer Cycles

• Consider an Application $a$ that owns a `std::shared_ptr<Screen>` $s$. If $s$ owns a `std::shared_ptr<Application>` to $a$, then $a$ cannot be destroyed without manually destroying $s$
  
  - Why? Consider the diagram below. Application $a$ is referenced twice (by the rest of the program and by $s$)
  
  - When we destroy the rest of the program, Application $a$ is not destroyed because its reference counter decreases from 2 to 1 (so the counter does not reach 0)
Avoid Shared Pointer Cycles

• It is very common for a Screen to want to reference the Application that owns it
  – We can have this behavior and avoid memory leaks by having the screen own a raw pointer to the application
  – This is safe to do because the Application owns the Screen, but the Screen does not own the application (shared pointers shown ownership)

• This pattern will be useful when we talk about GameWorlds, GameObjects and Components as well!
In Summary…

• Unique/shared pointers make memory management easier

• Please don’t have memory leaks in your handin code
Warmup 1 is released! Good luck!