LECTURE 1
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 RISD 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 CS195n/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
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 CIT 316
  - 1-1.5 hours of lecture
  - 20 minutes of playtesting
- Design Checks: Saturday and Sunday
- TA Hours: Monday and Tuesday
- Assignments due every Tuesday
To take this class, you must...

- Have taken a software engineering course (in order of usefulness):
  - CS123, CS195n, CS32, CS33
- Have knowledge of OpenGL, meaning you have either:
  - Taken CS123
  - Completed the winter assignment
- Be comfortable with object-oriented design
- Be comfortable with programming large C++ projects
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 with primary, secondary, and tertiary requirements
  – Handin due every Tuesday at 11:59:59 PM
Requirements include...

• Global requirements
• Weekly checkpoint requirements (split up into primary, secondary, and tertiary requirements based on importance)
  – Engine requirements which go into src/engine
  – Game requirements which go into src/<projectName>
• Playtesting other students’ games
  – Help find bugs, give feedback on gameplay
  – Only for weeks with “real” gameplay
• Design checks are MANDATORY
  – Sign up using cs1972_signup <project>
  – High level conceptual questions
Grades

• Finishing all requirements – A
• Finishing primary and secondary requirements – B
• Finishing primary requirements – C
• Primary requirements not done - incomplete
I got an incomplete, now what?

- You will get 5 retries (late weeks) to use throughout this semester
  - This gives you one more week to work on your project
- Successful retry can bump your C on a project to an A, or help you turn that incomplete into a grade
- 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 is an average of your grades for all your projects
• Must hand in a working version of all primary checkpoints to pass!
A word of warning...

- “Snowball of death”
- Start early!!!! You use your previous projects to make your next ones!
Expected hours

Number of hours

Warmup I | Warmup II | Minecraft I | Minecraft II | Minecraft III | Platformer I | Platformer II | Platformer III | Platformer IV

0 | 10 | 15 | 20 | 25 | 10 | 15 | 10 | 15
ABOUT REGISTRATION

Introduction
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
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(float seconds)
• 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 (FPS) in all projects
Draws

• General contract:
  – \texttt{void onDraw(Graphics *g);};
  – Convert game state into viewable form
  – Everyone makes their own \texttt{Graphics} object to abstract out common drawing methods

• \textbf{MUST BE FREE OF SIDE EFFECTS!}
  – Two subsequent draw calls should produce identical results if the game state has not changed
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

- The **Application** is the top level class that represents an instance of a game
  - You will implement this in Warmup1
- The Application follows this interface
  - support code (view.cpp) needs to know about application to call these methods on the application

```cpp
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);
};
```
Basic Engine Architecture (Common Engine)

SCREEN 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: 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
  – onTick
  – onDraw
  – Input event methods
Keeping track of Screens

- **Simplest way:**
  - Single Screen in Application at a time
  - Screens call `setScreen()` on Application

- **Alternative way:**
  - List of Screens maintained by the Application
  - Topmost Screen gets events (most of the time) or multiple screens can get these events (special case)
Application Management

QUESTIONS?
LECTURE 1
Camera and Graphics
(Common Engine)
FIRST PERSON CAMERA
Coordinate systems

• Different game engines define 3D coordinate systems differently
  – Most of you will probably use the traditional right-handed coordinate system
• “Horizontal plane”
  – Plane parallel to the ground (the xz-plane)
• “Up-axis”
  – Axis perpendicular to horizontal plane (the y-axis)
Cameras in a 3D Space

- Physical camera will render a “film” – a 2D representation of the 3D space
- For virtual cameras, goal is similar
- How virtual cameras render
  - Squash view frustum onto viewplane
Object Space

- First step of rendering is describing objects in our world
- Object space has unit spheres, cubes, etc.
- Shapes centered at origin
World Space

- Second step is describing sizing and positioning of objects relative to one another
- World space is space where objects relatively sized and positioned

Objects are typically created in their local spaces. We need to bring them into the common world space, via a series of affine transforms (translation, rotation and scaling).
Camera Space

• Third step is describing position of objects relative to the camera
• Camera space where camera is placed at origin and all objects are placed relative to camera
• The reason why we don’t use this as our reference space is because the camera is constantly moving
• We want this as intermediate space before...
Screen Space

• Last step, squash 3D space onto a 2D plane
Matrix Transformations

• **Model matrix**
  – Converts the vertex from object local coordinates to world coordinates
  – Different per object

• **View matrix**
  – Converts world coordinates to camera coordinates
  – The same for all object and based on camera location and orientation

• **Projection matrix**
  – Has parameters (aspect ratio, field of view), usually only changes on window resize

For more information on matrices, see [https://cs.brown.edu/courses/cs123/docs/helpsessions/linearalgebra.pdf](https://cs.brown.edu/courses/cs123/docs/helpsessions/linearalgebra.pdf)
Our Camera Class

• A bit different from CS123’s camera class – check out the winter assignment
Yaw/Pitch

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

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

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

- Position of camera in world space...
- In order to achieve first person
  - Make camera position same as player position
  - Update camera position to make the same as player position
Horizontal player movement

- Simple trigonometry for horizontal movement
  - Camera is rotated (yawed) around the up-axis at some angle, $\theta$
  - Use angle to get vector of forward velocity (direction player is facing) in horizontal plane
    \[
    \begin{align*}
    vel.x &= \cos(\theta) \\
    vel.z &= \sin(\theta)
    \end{align*}
    \]
  - Multiply velocity by time since last tick and add to position
    \[p_{\text{horizontal}} += vel \cdot t\]
- Strafing
  - Same as above, but use angle 90° left or right from the player’s facing direction
Vertical player movement

• Acceleration due to gravity
  – $g = \text{(a reasonable negative constant)}$
  – $v_{up} += g \cdot t$
  – $p_{up} += v_{up} \cdot t$

• Collision with ground
  – After previous step: $p_{up} = \max(0, p_{up})$
  – $v_{up} = 0$
First Person Camera

QUESTIONS?
Camera and Graphics (Common Engine)

BASIC GRAPHICS
Motivation

• Certain graphics calls are common to many games
  – Loading a texture
  – Drawing a texture
  – Drawing a primitive shape

• We can store all of our textures and shapes 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

• Using 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

- See winter assignment step 3 for details.

1. Generate a VBO and VAO.
2. Bind the VBO and VAO.
3. Define the data for the primitive.
4. Send the data to the bound VBO with `glBufferData`.
5. Set up the VAO with `glEnableVertexAttribArray` and `glVertexAttribPointer`.
6. Clean up by unbinding the VBO and VAO.
Shape Wrapper

• If you have taken CS123 this should be familiar
• Encapsulates OpenGL calls for drawing geometry
• Sample Contract similar to CS123
• Take a look at the support code that we provide for this — find in /course/cs1972/asgn/datatype

```cpp
Class Shape {
public:
    Shape();
    virtual ~Shape();

    void setVertexData(GLfloat *data, GLsizei size, int numVertices);

    void setAttribute(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, size_t pointer);

    void draw();

private:
    GLuint m_vbo;   // ID of the VBO
    GLuint m_vao;   // ID of the VAO
    int m_numVerts; // Number of vertices to draw
};
```
Shape Wrapper

- **Constructor:** Generate VBO/VAO
- **Destructor:** Delete VBO/VAO
- **setVertexData:**
  - Store numverts
  - Send the data to the bound VBO with `glBufferData`. Make sure to bind and unbind the VBO
- **setAttribute:**
  - Set up the VAO with `glEnableVertexAttribArray` and `glVertexAttribPointer`. Make sure to bind and unbind the VAO/VBO
- **draw:**
  - Use `glDrawArrays` to draw geometry
  - Make sure to bind and unbind the VAO
- **setVertexData/setAttribute** should be done once per primitive
- **Draw** should be called when you want the shape to draw
Texture wrapper

- Take a look at the support code that we provide for this – find in /course/cs1972/asgn/datatype/Texture, /course/cs1972/asgn/datatype/Texture2D,
- You will probably be using the Texture2D class
  - Relevant methods are Texture2D::bind, Texture2D::unbind which simply bind and unbind the texture to GL_TEXTURE_2D
Graphics Object

• For this week, we require you to do texture loading, storing, drawing, and deleting in your graphics object
• We also require you to have shape storing, drawing, and deleting methods
• Important thing is that you are not re-creating textures, shapes each draw call
• Tip: you will probably want to store textures and shapes in a map<string, Texture/Shape>, or map<string, GLuint>
  – Depending on whether using wrappers or not
Graphics Object

• Use it for any method you think you might want to use somewhere else in your code or in another game

• Think about how to further encapsulate OpenGL with wrappers for shapes, VAO, VBO, etc
  – See /course/cs1972/asgn/datatype
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 – glm
  – 2,3,4 dimensional vectors and matrices
  – Tons of math – see online documentation

• QRC files
  – Allows for easy access of external resources
  – Use for texture/shader loading this week
Support Code Overview

• Utility
  – CommonIncludes.h
    • Includes glm, glew, iostream
    • Include this anywhere you need ogl/glm
  – CylinderData.h
    • Hardcoded Cylinder data for drawing
  – ResourceLoader.h/.cpp
    • Same as CS123, used to load shaders

• Glew
  – The OpenGL Extension Wrangler Library
  – Gives access to OpenGL calls with cross platform support
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()
  – initializeGL() – can’t make OGL calls until here
  – resizeGL(int x, int y) – call glViewport(), inform camera

• Make Application a separate class from View!
  – Put instance of Application class in View, so that you can pass events on to Application
More Things About Support Code

• You **do not** have to mess with the shader classes at all in this course (unless you want to)

• For more information about the support code and more tips for Warmup1 check out the winter assignment
Drawing Shapes

Without Transforms

• Calculate coordinates of each vertex for each time you want to draw a different polygon
• Update the VBO every time you want to draw
• Profit!
• Copy/paste code and replace vertices 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
Drawing a full screen quad

- Quad that spans all of OpenGL screenspace
  - (-1, -1) lower left, (1, 1) upper right
  - Set model, view, projection matrices in shader to identity (no transformation), and use the following vertex data

- Good for menu screens!
  - Warmup1 requires you to have at least 2 screens

```cpp
float quadData[] = {
  -1, -1, 0, // position
  0, 0, 1, // normal
  0, 1, // texture
  1, -1, 0, // position
  0, 0, 1, // normal
  1, 1, // texture
  -1, 1, 0, // position
  0, 0, 1, // normal
  0, 0, // texture
  1, -1, 0, // position
  0, 0, 1, // normal
  1, 1, // texture
  1, 1, 0, // position
  0, 0, 1, // normal
  1, 0, // texture
  -1, 1, 0, // position
  0, 0, 1, // normal
  0, 0, // texture
};
```
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/qtc core.html
Qt vs. 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
class MyClass {

private:
  bool check;  int x, y;

public:
  // Usually prints garbage
  MyClass() {
    if (check) cout << x + y << endl;
  }
  // Always prints garbage
  MyClass() : check(true), x(3) {
    if (check) cout << x + y << endl;
  }
  // Always prints 8
  MyClass() : check(true), x(3), y(5) {
    if (check) cout << x + y << endl;
  }
};

Remember to initialize any and all instance variables!
- Bugs will make no sense
- Things will fly off the screen
- You will get frustrated
- Use an initializer list or in-class member initializers
  - Member variables are initialized once you get to the constructor body
C++ Tip of the Week

SMART POINTERS
Raw pointers

• Problems:
  – Declaration doesn’t indicate single object or array
  – Declaration doesn’t indicate who owns the object (i.e., who destroys it)
  – If you know that you must delete it, doesn’t tell you how
  – If delete is that way to go, is it an array?
  – Must destroy exactly once
  – Dangling pointers
Smart Pointers

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

• 3 types in modern cpp (c++14)
  – Std::unique_ptr; std::shared_ptr; std::weak_ptr
Unique Pointers

- In general the one to use
- In general, same size as raw pointers and perform the exact same instructions
- Used for exclusive ownership
  - Non-null std::unique_ptr always owns what it points to
  - Moving a pointer (more on this later) transfers ownership
  - Copying is not allowed
- Std::unique_ptr<T[]> for array objects. Almost never used as std::array, std::vector, std::string are preferred
Unique Pointers example

- Basic use
- Destructor is called automatically – don’t need to call “delete”

```cpp
#include <memory>

class Chunk{
    int size = 32;
};

int main()
{
    unique_ptr<Chunk> c;
    c.reset(new Chunk);
    c->size;
    return 0;
}
```
Unique Pointers example

• Passing unique pointers by value
  – Used to pass ownership

• This will give you a compile error - trying to copy the pointer

• How do you call it?
  – `std::move`
  – “test(std::move(c))”
  – `unique_ptr` in test function now owns the resource, and `unique_ptr` in main does not

```cpp
#include <memory>
class Chunk {...}

void test(unique_ptr<Chunk> c){return}

int main()
{
    unique_ptr<Chunk> c;
    c.reset(new Chunk);
    test(c);
    return 0;
}
```
Include <memory>
class Chunk{...
    virtual ~Chunk(){
        std::cout <<"I have been deleted" <<std::endl;
    }
}

void test2(unique_ptr<Chunk> c){return}

int main()
{
    unique_ptr<Chunk> c;
    c.reset(new Chunk);
    test2(std::move(c));
    std::cout << c.get() << std::endl;
    return 0;
}
Unique Pointers example

- **Passing unique pointers by reference**
  - May or may not change ownership (up to you)
- **Prints**
  - Address
  - I have been deleted
  - null

```cpp
#include <memory>

class Chunk{...
    virtual ~Chunk(){std::cout <<"I have been deleted" <<std::endl;
    }

void test2(unique_ptr<Chunk> &c){return;}
void test3(unique_ptr<Chunk> &c){
    auto a = std::move(c);return;}

int main()
{
    unique_ptr<Chunk> c;
    c.reset(new Chunk);
    test2(c);
    std::cout << c.get() << std::endl;
    test3(c);
    std::cout << c.get() << std::endl;
    return 0;
}
```
Unique Pointers example

- Pass Pointers by const reference
  - Other object can not take ownership
  - Compile error

```
#include <memory>

class Chunk {...
  virtual ~Chunk() { std::cout << "I have been deleted" << std::endl; }
};

void test4(unique_ptr<Chunk> const &c) {
  auto a = std::move(c);
  return;
}

int main() {
  unique_ptr<Chunk> c;
  c.reset(new Chunk);
  test4(c);
  return 0;
}
```
In Summary...

• Try to use unique pointers, but pass by reference
  - Otherwise you may end up with a null reference if you’ve accidentally changed the ownership
LECTURE 1

C++ Anti-Tip of the Week
Backwards Iteration is Best!

```cpp
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!