Class 2 Announcements
Text

• To draw text
  – camera->setUI(true)
    • 1 unit = 1 pixel
    • Positions are pixel offset from bottom left of the screen
  – g->setMaterial(...)
  – g->drawText(“Hello, world”, 100)

• To increase font resolution
  – Change fontResolution variable in engine/graphics/Font.h

Each character is 100 pixels tall
Handins

• Since we have had some issues with FastX, you do not need to check whether your handin compiles on the department machines

• All you need to do now is write what OS, Qt version, and compiler you are using in your README
Didn’t Complete Warmup1?

• No problem — you have retries!
• Don’t let the snowball begin week 1
  — Try to do your retry AND Warmup2 this week!
Collaboration Policy

• Remember to sign the collaboration policy on the website

• Pretty vanilla collaboration policy
  – **No** sharing code
  – **No** looking at each other’s code
  – **Can** discuss design
Warmup2 - Your first full game!

• You’ll have your first full 3D game after this week!
• Gameplay options are actually pretty diverse
  – More on this later
• Have some fun!
QUESTIONS?
Class 2

Third Person Camera
Third-Person Camera

THE THIRD PERSON CAMERA
First Person is easy

- Field-of-view is limited
- Actions are (almost) always happening in the direction the player is looking
- It’s how we see the real world
Third Person is tricky

• Field of view is ambiguous – player can often see:
  – Behind themselves
  – Around corners
  – Through walls

• Player can perform actions without turning
  – Fighting sequences

• We don’t see the real world this way
What works best?

Player controls the camera?  Camera controls itself?
Combine the two?

- Camera automatically turns to keep player in focus as well as possible
- Player can manually change the camera if they want a particular camera angle
The Simplest Solution

• Take the first person camera
• Translate the eye back along the look vector
• Pros:
  – Easy to toggle between 1\textsuperscript{st} and 3\textsuperscript{rd} person cameras
  – Easy to change zoom level by scaling translation
• Cons:
  – Awkward camera controls (pitch and yaw don’t always feel quite right)
  – Sometimes clips through walls (can use raycasting to circumvent this, but we’ll get to that another day!)
Third Person Camera

QUESTIONS?
CLASS 2

Game World
Games are busy…

• All games involve a number of game objects
  – May be many different types of objects
  – May be many instances of the same type of object
  – Maybe both of the above

• These objects exist in their own universe
  – If our entire universe is a game, you’re a game object
Games are busy...

• We need to take the burden of representing and organizing these objects off the game code
  - Otherwise, have to re-implement for every game we build
High Level Representation

The **GameObjects**
- Small collections of functionality
- Hold their **own** logic and state

The **GameWorld**
- The overarching collection of **GameObjects**
- Responsible for global logic and facilitating interactions between **GameObjects**
 QUESTIONS?
Game World

GAME OBJECTS
What is a GameObject?

- Environment
- Player
- Enemies
- Much more!

- How do we implement them?
Hierarchical GameObject design

• Consider a simple game with:
  – Player
  – Enemies
Hierarchical GameObject design

- They both move and have health bars
- Some behavior specific to Player
- Some behavior specific to enemies
Hierarchical GameObject design

- Let's add a new type of object: Destructible Terrain
  - Has a health bar
  - Doesn't move
- Time to abstract out some more

```
GameObject
  • Health bar

Units
  • Position

Terrain
  • Terrain shape

Player
  • Key controls

Enemy
  • AI
```
Hierarchical GameObject design

• Let’s add Invincible Enemies
  – No health bar
  – They move
• Now we reach a problem
• Where do our invincible enemies fit in?
Hierarchical GameObject design

- Attempt 1: Can make a separate class and re-implement moving and AI
Hierarchical GameObject design

- Attempt 2: Can subclass enemy and add code to hide the health bar
Hierarchical GameObject design

• Both not ideal
  – Attempt 1: Re-implementing code
  – Attempt 2: Unused superclass functionality in subclass (not good practice)

• With hierarchical design, often have to make these lose-lose design decisions
Solution

- Component-based design
Solution

- **GameObject**s are just lists of components
- The components implement all relevant functionality
- The appearance and logic of each object is defined by its components
  - objects can be any combination of components
- Making new objects is as easy as adding new components

---

**GameObject**
- List of Components

**HealthComponent**
- Health bar

**PhysicsComponent**
- Updates position

**DrawComponent**
- Has a Shape, Material

**AICComponent**
- Makes decisions

**PlayerControlComponent**
- Responds to key presses
GameObject Contract

• GameObjects need to do the following:
  – Add a component
  – Remove a component
  – Get a component
class GameObject {
    public:
    void addComponent(...);
    void removeComponent(...);
    Component getComponent(...);

    private:
    Container<Component> m_components;
};
Component Contract

• Nearly all Components need to update
  – onTick
  – Where most logic is implemented

• Some Components respond to other events
  – For example onDraw

• Components might want to access parent GameObject
  – So that they can talk to other components of the same GameObject
Component Contract

class Component {
public:
    void onTick(float seconds);
    // more events (possibly in subclasses)

private:
    GameObject *m_gameObject;
}

QUESTIONS?
Class 2

Cylinder-Cylinder Collisions
Collisions I

CYLINDER-CYLINDER COLLISIONS
Parts of a collision

• Detection
  – Are two shapes overlapping?

• Resolution
  – Make them not overlapping anymore

• Response
  – Make them bounce off each other in some believable way
Why do we need it?

• (Almost) every 3D game uses it

• Even last week, you did this with the floor
  – Is pos.y < 0? (detection)
  – If so, make pos.y = 0 (resolution)
  – Set the player’s y velocity to 0 (response)
Cylinders

- Cylinders make great collision shapes
  - People are kind of cylinders
  - The math is pretty easy
  - Turning in place doesn’t change your collision shape
Concept

• Separate 3D problem into 2D and 1D problems
  – $2 + 1 = 3$

• Overlapping if both:
  – Bases overlap in xz plane
  – Heights overlap on y axis

• Easy if your cylinder is represented by a point (bottom center) and dimension (radius, height)
Concept

• Need to find minimum translation vector (MTV)
  – Minimum Translation Vector — shortest possible translation to get two shapes out of collision
  – With respect to one of the shapes in collision
Concept

• Either translate in xz plane or in the y direction
  – Only 2 possible MTV’s
  – Pick the one that is shorter

• Translate red out by $\frac{1}{2} m_{tv}$ and blue out by $-\frac{1}{2} m_{tv}$
  – If your engine supports immovable game objects, the movable game object is translated out by the entire MTV
Circle Math

Two circles are overlapping if and only if:

- \((\text{blue.pos} - \text{red.pos}).\text{length()} < \text{blue.radius} + \text{red.radius}\)

Avoid square root by squaring expression!

- \((\text{blue.pos} - \text{red.pos}).\text{lengthSquared()} < (\text{blue.radius} + \text{red.radius})^2\)
Computing Circle MTV

MTV (in the direction of red):

- \( \text{len} = (\text{blue.pos} - \text{red.pos}).\text{length()} \)
- \( \frac{(\text{red.pos} - \text{blue.pos})}{\text{len}} \times ((\text{blue.radius} + \text{red.radius}) - \text{len}) \)
Two 1D line segments are overlapping if and only if both of the following are true:

- $\text{blue.min} < \text{red.max}$
- $\text{red.min} < \text{blue.max}$
Computing Line MTV

```c
float intervalMTV(Interval a, Interval b) {
    float aRight = b.max - a.min;
    float aLeft = a.max - b.min;
    if (aLeft < 0 || aRight < 0) {
        return -1;
    }
    if (aRight < aLeft) {
        return aRight;
    } else {
        return -aLeft;
    }
}
```
Collision Response

• If objects collide, they should do something
  – Minimally, translate by $\frac{1}{2}$ the MTV each
    • Adds up to 1 full MTV!
  – In almost all cases, do game-specific logic

• Example: bullet collides with player
  – Player takes damage
  – Player is moved back some by the force of the bullet
  – Bullet is destroyed
Class 2 Systems
MOTIVATION

Systems
Example: Collision Logic

• We know how to detect if cylinders are overlapping
• We have a bunch of GameObjects
• How do we make them collide with each other?
Example: Collision Logic

• We could put collision logic in the `GameWorld`
  – `GameWorld` checks if each pair of collidable `GameObject`s is colliding
  – `GameWorld` tells `GameObject`s that they there are colliding
  – Components of `GameObject`s respond appropriately

• Sounds pretty good
Global Logic

- What if your GameObjects play sounds?
- Should we put sound logic in the GameWorld too?
  - GameWorld tells each GameObject that can play a sound to do so
Global Logic

• What if other GameObjects require more global logic?
• We would have to add all of our logic to the GameWorld class
• Before long, our GameWorld can get pretty bloated ...
Systems

• Introducing **Systems** ...
• **Systems** implement global logic
  – Each **System** stores a list of interested objects (usually **GameObjects** or **Components**) and calls relevant methods on each of them
• Other examples
  – **DrawSystem** calls `draw(Graphics *g)` on its drawable components
  – **CollisionSystem** checks collisions + calls `collide(glm::vec3 mtv)` on collision components
  – **SoundSystem** calls `playSound(...)` on sound components
IMPLEMENTATION
Storing **GameObjects vs. Components**

- You could have your **Systems** keep track of either a list of **GameObjects** or a list of **Components**—it's up to you
  - Storing **GameObjects** makes it easier to access multiple **Components** of the **GameObject** and reference **GameObject** specific state
  - Storing **Components** avoids potential type headaches and can give you performance benefits
System Contract

class System {
public:
    void onTick(float seconds) {
        for(GameObject/Component obj : m_objects) {
            // Update objects
        }
    }

    // more events (possibly in subclasses)

private:
    Container<GameObject/Component> m_objects;
}
What about the GameWorld?

• **GameWorld** should hold Systems as well as GameObjects

• With **GameObject/Components**:
  – The appearance and logic of each object is defined by its components

• With **GameWorld/System**s
  – The global logic and interactions between objects in the world are defined by its systems
class GameWorld {
public:
    void tick(float seconds);
    void draw(Graphics *g);
    //...

private:
    Container<GameObject> m_objects;
    Container<System> m_systems;
}
QUESTIONS?
Class 2

Handling Input
Last time...

• Right now, you might be handling input like this:

```cpp
void onKeyPressed(QKeyEvent *event) {
    if (event->key() == Qt::Key_W) //move up
    if (event->key() == Qt::Key_S) //move down
        ...
}
```
The Problem

• Qt events may not act like you would expect
• The `onKeyPressed` event fires rapidly for the duration a key is held
  – Think of when you press down a letter key in Microsoft Word
The Result

• Movement happens in short bursts
  – We move a little bit every time a key event fires from the system
• Result: jerky movement—not ideal!
• What we want is something smooth and continuous
The Solution

• A key has two states: pressed and not pressed
• This is represented in the event system by KeyPressed and KeyReleased events
The Solution

• Instead of moving on every single KeyPressed event, just pay attention to first one
• Until the KeyReleased event, move the player every tick
• Keep track of the current state of each key in some sort of data structure of Booleans
  – For example, Map<int, bool>
But wait, there’s more!

• In general, it’s useful to access the current state of input ...
  – Keys that are pressed
  – Mouse buttons that are pressed
  – Mouse position
• Not just when the current state changes
• **Input** object that stores all of this information?
• Up to you—be creative!
QUESTIONS?
CLASS 2

Tips for Warmup 2
Warmup 2 Design

• This week, design is everything

• Many ways to approach GameWorld, System, GameObject, Component design
  – Even with the contracts we gave you

• For example …
  – Systems can store GameObjects or their Components

• Plan out your design, and talk it through with TAs
More Design Tips

- **GameWorld and GameObject**
  - Should be defined engine side
  - Should not be subclassed game side or engine side

- **System and Component**
  - Should be defined engine side
  - Can be subclassed game side or engine side
Your First Components (Suggested)

- **DrawableComponent**
  - Holds shape, material
- **TransformComponent**
  - Holds position, size of object
- **CollisionComponent**
  - Holds collision cylinder
- **PlayerControlComponent**
  - Responds to player input
- **Player/EnemyResponseComponent**
  - Does something in response to collision
Your First Systems (Suggested)

- **TickSystem**
  - Ticks the objects that it holds

- **DrawSystem**
  - Draws the objects that it holds

- **CollisionSystem**
  - Checks for collisions between objects
  - Notifies objects that they have collided
Destroying Game Objects

• Your Warmup 2 game needs to have a reset feature
• You must clear the gameworld when you reset
• Think about what would happen if a gameobject $G$ managed by a `shared_ptr` called a method `destroySelf()`, which removes all `shared_ptr` references to $G$ in the gameworld
  – The number of references to the shared pointer goes to 0, and then the shared pointer is destroyed!
  – But we are inside of one of $G$’s methods, so we get a segfault!
• Think about what you could do to bypass this problem
Entity Component System Typemap

• Given a gameobject, what is the best way to access its components?
• We could have some function like Component getComponent(std::string s)
• This is okay, but if we wanted to access any methods of a CollisionComponent that are not virtual functions in the abstract Component class, we would need to cast the Component to the CollisionComponent class
Entity Component System Typemap

• We can use some C++ tricks to create a typemap
• There is a typemap implementation in src/engine/utils/TypeMap.h
• Look at this article for an explanation of the code!
  — Feel free to ask about how the typemap code works!
Entity Component System Typemap

• You can use the typemap in your GameObjects like this (assuming we have a TypeMap<std::shared_ptr<Component>> called m_components)

```cpp
template <typename Comp>
void addComponent(std::shared_ptr<Comp> &&c) {
    m_components.put<Comp>(std::forward<std::shared_ptr<Comp>>(c));
}

template <typename Comp>
void removeComponent() {
    m_components.remove<Comp>();
}

template <typename Comp>
std::shared_ptr<Comp> getComponent() {
    auto it = m_components.find<Comp>();
    return std::static_pointer_cast<Comp>(it->second);
}
```
Entity Component System Typemap

• And then we can ask a gameobject for its components like this...
• No casting required! (It’s covered by the GameObject class)
• We can add extra assert statements to make sure our typemap is consistent

\[
glm::vec3 \text{ position} = \text{m_gameobject->getComponent<TransformComponent>()->getPos();}
\]
Cylinders

• What are they good for?
• Absolutely nothing
Tips for Warmup2

QUESTIONS?
CLASS 2

C++ Tip of the Week
C++ Tip of the Week

FORWARD DECLARATIONS
Forward Declarations

• What is declaration (as opposed to definition)?
  – Just enough information to tell the compiler ‘this exists’
    • For a function, it would be the type signature:
      ```
      int add(int a, int b);
      ```
    • For a class, it’s just the name:
      ```
      class Number;
      ```
  
• When you `#include` a header, you’re defining that class
Forward Declarations

• When do we actually need to `#include` a class?
  – If this class extends that class
  – If it has a non-pointer member variable to that class

• That means we don’t need to `#include` classes of:
  – Pointer member variables
  – Function arguments & return types (pointer or non-pointer)
Forward Declarations

• When should we use forward declarations?
  – For every class, in every header file, that you possibly can
  – There is no benefit to including a whole class when you only need the declaration
  – You can put the `#include` in the `.cpp` file instead
Benefits of Forward Declaration

– Fewer circular dependencies
  • These happen when two classes rely on each other, and both attempt to define the other by `#including` headers
  • If the link is indirect, it will take much longer to track

– Significantly reduced build times
  • Every time you `#include` a header, it also then `#includes` all the headers `#included` by that header, and so on
  • Game engines are huge, and this problem multiplies per file
Next Week...
Class 2

Good luck on Warmup 2!