LECTURE 2
Announcements
Comments on Warmup1

• Test on the department machines before you hand in!
  – Sometimes things run differently from your personal machines
Incomplete on Warmup1?

- No problem – you have 5 retries!
  - Due in 1 week (same due date/time as warmup2)

- If you received a B or a C, you can use a retry to bump up your grade
  - But you might consider saving it for later

- Don’t let the snowball begin week 1
  - Do your retry AND warmup2 this week!
  - Don’t just do the entire class one week behind – it will come back to haunt you
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!

• First playtesting session after lecture today
Playtesting

- Play 4-5 other student’s games
  - Part of your warm up 1 req’s
- Play as a player:
  - What did they do that you like?
  - What did they do that you didn’t like?
- Play as an engineer:
  - Can you find any bugs? Can you reproduce them?
  - Do you understand how their systems work?
- Quality > quantity for feedback
  - “I like X” is nice, but “I like that X uses Y to do Z” is much better
  - “It crashed” is useless, but “It crashed when I did X after doing Y, but didn’t when I did X after doing Z” is SO helpful
Course Logistics

• We’re going to hand out collab policies now
• Once you give us a collab policy, we will give you an override code to register for the class on Banner
• To find the course on Banner, uncheck “Exclude Independent Study” and search for 1972 under CSCI
QUESTIONS?
LECTURE 2
Third-Person Camera (Common Engine)
Third-Person Camera (Common Engine)

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 1st and 3rd person cameras
  – Easy to change zoom level by scaling translation
• Cons:
  – Awkward camera controls (pitch and yaw don’t feel quite right)
  – Sometimes clips through walls
• We recommend supporting one camera that has both modes built directly in
Third Person Camera

QUESTIONS?
LECTURE 2
Game World Representation
(Common Engine)
Game World Representation (Common Engine)

MOTIVATION
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
- They exist in their own universe
  - If our entire universe is a game, you’re just an object
- We need to take the burden of representing and organizing these objects off the game code
High Level Representation

The Game Objects - “Entities”
• Individual units of functionality
• Responsible for their own logic and state
• Don’t know they’re a part of a game

The Game “World”
• The overarching collection of entities and state
• Responsible for global logic and facilitating entity logic
• Represents the boundary between program and game
World Responsibilities

- Represents all “entities” in a single space
  - “Owns” game coordinate system
- Centralizes entity management
  - Maintains list, add/remove via queues, iterating over copy, or a fancy backwards iteration
    - Backwards iteration and iteration over copy to prevent concurrent modification
  - Passes ticks and draws to entities
- Handles global logic
  - Anything beyond the scope of a single entity
  - E.g. providing collision detection callbacks to entities and handling mouse/key/other input events
Entities

• Single logical “object” within the game
  – Stores all state relevant to itself, e.g. drawing information, collision shape, HP, attack cooldown, etc…

• Hierarchical representation
  – High-level subclasses in engine, low-level in game
  – Each additional layer adds a specific set of functionalities

• Receives events from World
  – More than just tick+draw!

- Entity
  - Background
  - Damagable
  - Bullet
  - Enemy
  - Player
  - Boss
Entity responsibilities

- May draw, may not
  - Spawners, timers, force fields don’t
- May use ticks, may not
  - Static environment, background don’t
- Most will probably do both though
  - Player, enemies, items, pickups, and many more
- Player input?
  - Better handled by World (the boundary between program and game)
- Determine collisions with other Entities?
  - Better handled by World (the collection of all entities)
- Respond to collisions with other Entities?
  - Handled by Entities, allows each Entity to uniquely specify how it responds
What about the environment?

• This week, the environment is just the $y=0$ plane
  – For minecraft, it will be a bunch of cubes
  – For platformer, it will be a bunch of polygons
• How the environment is represented determines how entities must be represented
  – For warmup, cylinders work great!
  – For minecraft, boxes work great!
  – For platformer, ellipsoids work great!
• We can even mix collision systems – more about that in upcoming weeks
Multi-pass logic

• Ticking and drawing entities in the wrong order leads to undesirable behavior
  – Drawing background over everything else (only really applies in 2D)
  – Entities removing themselves during collision detection
• World can selectively update state in order
  – E.g. tick all entities so they update position, *then* check for collisions
  – Can even specify draw ordering with an interface
Game World Representation (Common Engine) - Motivation

QUESTIONS?
Game World Representation (Common Engine)

IMPLEMENTATION
The Entity class

• (Almost) every Entity minimally needs to be able to:
  – Tick (exist in time)
  – Draw (exist in space)
• Since we’re focusing on collisions, it also makes sense for every Entity to have:
  – Position (as a 3D vector)
  – Dimensions (as a 3D vector)
    • This will allow us to determine several types of bounding shapes
  – Other physics components?
    • Velocity, acceleration are incredibly useful and trivial to implement
    • Mass, force, and impulse are a bit trickier (we won’t be implementing them in this class)
The Entity class

```cpp
class Entity {
public:
    // Entities must at least have a position and dimensions
    Entity(vec3 pos = vec3(0,0,0), vec3 dim = vec3(0,0,0));
    // Concrete but overridable tick and collisions, abstract draw
    virtual void onTick(float seconds);
    virtual void onDraw(Graphics *g) = 0;
    // Collision methods (more to be added later...)

protected:
    vec3 _position, _dimensions;
    // Other physics values like velocity, acceleration, etc...
};
```
You will implement an abstract `World` class in your engine, and will subclass it in your game.
The world maintains a collection of all Entities.
Every tick, the world:
  - Ticks all entities (engine-side)
  - Does game-specific logic (game side)
Every draw, the world:
  - Updates the transformation matrices in the shader (engine-side)
  - Draws all entities (engine-side)
Game code subclasses `world` and adds game-specific logic, including additional tick logic and event handling.
The World class

class World {
public:
    // Concrete but overridable tick and draw
    virtual void onTick(float seconds);
    virtual void onDraw(Graphics *g);
    // Abstract input events
    virtual void onDDDEEE(QDDDEEEvent *e) = 0;
protected:
    QList<Entity *> _entities;
};
The World class

```cpp
void World::onTick(float seconds) {
    foreach (Entity *e, _entities) {
        e->onTick(seconds);
    }
}

void World::onDraw(Graphics *g) {
    // update camera matrices
    // call helper function to send matrices to shader
    foreach (Entity *e, _entities) {
        e->onDraw(g);
    }
}
```
Game World Representation (Common Engine) - Implementation

QUESTIONS?
LECTURE 2
Collisions I
Collisions I (Common Engine)

CYLINDER-CYLINDER COLLISIONS
Components of a collision

• Non-continuous collision detection:
  – Detection
    • Are two shapes overlapping?
  – Resolution
    • Make them not overlapping anymore
  – Response
    • Make them bounce off each other in some believable way

• Continuous collision detection:
  – Analytic detection
    • Compute where two shapes will overlap
  – Translation
    • Translate them to exactly that point
  – Response
    • Same as above
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 posn.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 MTV
  – Minimum Translation Vector – shortest possible translation to get two shapes out of collision
  – With respect to one of the shapes in collision
• 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 $\frac{1}{2} mtv$ and blue out by $-\frac{1}{2} mtv$
  – If your engine supports immovable entities, the movable entity is translated out by the entire MTV
Two circles are overlapping iff:

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

Avoid square root by squaring expression!

- \((\text{blue.posn} - \text{red.posn}).\text{lengthSquared()} < (\text{blue.radius} + \text{red.radius})^2\)

MTV (in the direction of red):

- \(\text{len} = |\text{blue.posn} - \text{red.posn}|\)
- \(\frac{(\text{red.posn} - \text{blue.posn})}{\text{len}} \ast ((\text{blue.radius} + \text{red.radius}) - \text{len})\)
Two 1D line segments are overlapping iff both of the following are true:

- $\text{blue.min} < \text{red.max}$
- $\text{red.min} < \text{blue.max}$
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
Response

• When your objects collide, they should do something
  – Minimally, translate by $\frac{1}{2}$ the 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
public class Entity {
public:
    Cylinder getCylinder();
    void onCollide(Entity *e, Vector3 mtv);
}

• Engine gives you callbacks that let you know when an Entity is hit by another Entity
• Allows you to appropriately change the velocities of each entity involved
  – Proportional to the difference in velocities
  – Exact physically correct collisions for these types of collisions covered in cs1971, come to hours if interested
One last tip...

• Hold up your right hand and repeat after us
  – “I will not”
  – “check for or respond to collisions”
  – “in an Entity’s onTick() method”

• Tick all your entities, then collide all your entities in the World
  – “Multi-pass” logic we talked about before
  – Things will break if you don’t
LECTURE 2
Tips for Warmup 2
Player motion

• Set velocity while keys are held?
  – Too sudden
  – Can interact poorly with collisions
    • Not a huge deal this week, but will be in the future

• Apply acceleration while keys are held?
  – Asteroids style movement!
  – Still not quite realistic
  – Most things have a max speed
Player motion

```c
pos += 10; // jerky movement
vel = 10;  // constant velocity
pos += vel;
acc = 1;
vel += acc;
pos += vel; // smoother, but not perfect
```
Goal velocity

- `goalVelocity` set directly from arrow keys
- Gradually set `velocity` to `goalVelocity`
- By applying an acceleration
  \[- A = k(v_{goal} - v_{current})\]
Cylinders

• What are they good for?
• Absolutely nothing
QUESTIONS?

Tips for Warmup 2
LECTURE 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
LECTURE 2
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;
```
WARMUP1 PLAYTESTING!

To the Sunlab!