Lecture 9

Announcements
Initial M3 Feedback

• These all look really good!
• Lots of creative ideas this week
• You’re now done with all the standard projects
• Have fun playtesting!
Congratulations!

- No more of our engine requirements!
- From now on everything you work on is designed by you
- Good work so far!
Final Coding

• Time to start coding for your final project
• This week you’ll be doing the bulk of your engine code
• Don’t forget to set up a meeting with your mentor TA soon
  – You probably already know your mentor TA, if not, check your email
Don’t forget about old projects!

- Don’t forget, you need a working version of each milestone to get credit for the class
- …but focus on your final project
QUESTIONS?
About “Advanced Topics”

• These lectures *will not* give you all the information you need to implement your engine features
  – They are intended as a high-level overview

• You will be responsible for researching the topics yourself

• We will make it clear what the expectations are during final grading meetings
The Illusion

• All players are playing in real-time on the same machine
• But of course this isn’t possible
• We need to emulate this as much as possible
The Illusion

• What the player should see:
  – Consistent game state
  – Responsive controls
  – Difficult to cheat

• Things working against us:
  – Game state > bandwidth
  – Variable or high latency
  – Antagonistic users
Send the Entire World!

- Players take turns modifying the game world and pass it back and forth
- Works alright for turn-based games
- ...but usually it’s bad
  - RTS: there are a million units
  - FPS: there are a million players
  - Fighter: timing is crucial
Modeling the World

- If we’re sending everything, we’re modeling the world as a uniform chunk
  - But it really isn’t!
  - Composed of entities, only some of which need input from a player
- We need a better model to solve these problems
Send Commands

- Model the world as local and shared data
  - Share player information, powerups, etc
  - Don’t need to share static level data
- Each player sends the other all actions that alter shared game world
- “Deterministic P2P Lockstep”
- Problem: everything must evaluate the same
  - Or else there are desyncs
- Problem: have to wait for all the other players’ commands
  - So everyone is limited by laggiest player
One player is the authoritative server
  - Now we don’t have to wait for slow players, just the server
Other player is a “dumb terminal”
  - Sends all input to server
  - Server updates the world and sends it back
Problem: client has to wait for server to respond to perform even basic actions
Client-side Prediction

- Client responds to player input immediately
- When the server sends back the authoritative game state, client state is overwritten

![Diagram showing the client-side prediction process]
Rollback

• But the server just sent a state that was 100ms in the past!
• What if games have diverged since then?
  – For instance, both players think they’ve collected a single powerup
• We can’t just replace our game world or we’ll lose commands from the local player
  – Client has to roll back the world and integrate commands since the last known good state
Masking the Timewarp

- **Problem:** laggy players experience this jump often
- **Solution:** if the server usually sends states from 100ms ago, run the client 100ms behind
- **Turns a jumpy experience into a smooth, only slightly slow one**
  - Very useful if relative timing of commands is important
What about the server?

• **Without rollback:**
  – In an FPS, would need to lead shots because the server won’t register shot until after delay

• **With rollback:**
  – The target could be shot after they think they’ve taken cover
  – Or we could delay the server player as well…

• Need to think carefully about both technical requirements and game impacts of any networking model
Networking

IMPLEMENTATION
TCP: Transmission Control Protocol

• Abstracts over IP
• All packets are guaranteed to be received and in the correct order
• Good for sending important, permanent data (websites, databases, etc)
UDP: User Datagram Protocol

• A very thin shell around IP
• Much faster than TCP, but no guarantees about reception or order
• Good for information where only the most recent state matters (streaming, etc)
TCP vs UDP

• (Very) generally: action games use UDP and turn-based games use TCP
  – World state updates can be lost without worry, commands not so much
• Can potentially combine them
  – TCP sends important data, UDP sends timely data
• Best choice varies by project
  – (for naïve version, TCP is fine)
Java Sockets

- Very good for most purposes
- Read and write objects to sockets
- UDP is deprecated for sockets; for UDP use DatagramSocket
Settings Up Sockets

- Open a connection on a port
- Open an input/output stream from the socket
- Read and write to the streams (which use the socket’s protocol)
- Close the streams and sockets

```java
String host = "127.0.0.1";
int port = 10800;
Socket out = new Socket(host, port);
ObjectOutputStream stream = new ObjectOutputStream(out.getOutputStream);
stream.writeObject("HelloWorld");
stream.close();
out.close();

String host = "127.0.0.1";
int port = 10800;
Socket in = new Socket(host, port);
ObjectInputStream stream = new ObjectInputStream(in.getInputStream());
String result = (String) stream.readObject();
stream.close();
in.close();
```
Edge Cases

• What if…
  – The client disconnects
  – The server dies
  – The client goes insane and sends gibberish
  – The client loses internet for 30 seconds
  – The client is malicious
  – The client changes IP address

• Handling errors well is vital to player experience
Elegant Disconnects

• Handle and respond to IO exceptions
  – Don’t just dump a stack trace
• Display informative status messages
• Send heartbeat packets every few seconds
  – Then respond if server/client hasn’t received a heartbeat in a while
• Never let the game continue to run in an unrecoverable state!
Java Serializable

• Need to represent an Object as data to send it over the network
• To enable serialization, just implement Serializable
• It’s an empty interface, so why isn’t everything serializable?
  – Not all objects are useful when serialized (Thread etc)
  – Don’t serialize passwords!
  – Need to provide compatibility
This is part of the compatibility contract

What if...
- You serialize a `MyGameState`
- You change the type of a field in `MyGameState`
- You load up the serialized data to the new class definition

`serialVersionUID` is a way of detecting this problem
- Serialization will die if the versions don’t match

```java
class MyGameState {
    int playerHealth;
    int playerX;
    float playerX;
    int playerY;
    float playerY;
}
```
Oh No Please Not Java Sockets!

- 3rd party libraries that handle the try/catch nonsense of using sockets, connecting, disconnecting
- Usually have their own serializations
- Most have both UDP and TCP
- Usually have onConnect() onDisconnect() onReceived() callbacks
- Kryonet is one example
To Library or Not to Library

• Pros:
  – Can simplify your code a lot
  – Typically handles serialization
  – Typically handles threading

• Cons:
  – You have less control over the networking model (typically forced into server-client)
  – You get 2/3 points

• Recommended library: Kryonet
Networking

QUESTIONS?
LECTURE 9
Open GL
What is OpenGL?

• **Not Swing**
  - `java.awt.swing` is what you have been using to draw your shapes/sprites
  - Instead of drawing shapes, you pass vertices to draw shapes
  - Loading images into memory is done by binding textures instead of storing `BufferedImage`

• **OpenGL will replace all Swing components in your rendering engine**
  - It’s much faster, and allows the use of shaders

• **Native to C++**
  - Using libraries such as JOGL or LWJGL, we can use OpenGL in Java
Integrating into your existing engine

• Create a Graphics interface
  – loadTexture(String path)
  – drawTexture(String name, Vec2f posn, Vec2f dim, Vec2f imagePosn, Vec2f imageDim)
  – Shape drawing!
    • drawRect(MyRectangle r)
    • drawCircle(MyCircle c)
    • Etc.

• Now your World and Screens can use a generic “Graphics” object
  – Game is agnostic of implementation
  – Game doesn’t need to know if Swing or OpenGL is being used
LWJGL

• Light Weight Java Game Library
• Wrapper library for OpenGL
• Gives you a window, event polling, access to drawing calls
• You can use OpenGL and an orthographic projection to draw your 2D game
• Gives you OpenGL as a set of static methods within static objects according to which version’s methods you want to use
  – Most functions that you will be interested in are in the static GL11 object, corresponding to OpenGL 1.1
  – Closely mirrors what using OpenGL is actually like in C++
public class Application extends SwingFrontEnd {
    public void onTick(long nanos)
    public void onDraw(Graphics2D g)

    public void onKeyPressed(KeyEvent evt)
    // more device and event types...
    public void onMouseDragged(MouseEvent evt)
}

Your job is to replace SwingFrontEnd with your own implementation
public void start() {
    try {
        Display.setDisplayMode(new DisplayMode(screenSize.x, screenSize.y));
        Display.create();
    } catch (LWJGLException e) {
        e.printStackTrace();
        System.exit(0);
    }
    initGL(); // set up your OpenGL environment
    long lastTick = Sys.getTime() * 1000000000 / Sys.getTimerResolution();
    while (!Display.isCloseRequested()) {
        pollInput(); // you will write this method to find out what events have happened, and call their methods
        if (Display.wasResized())
            onResize(new Vec2i(Display.getWidth(), Display.getHeight()));
        long time = Sys.getTime() * 1000000000 / Sys.getTimerResolution();
        long delta = time - lastTick;
        lastTick = time;
        onTick(time); // abstract method, overridden in Application
        onDraw(g); // abstract method, overridden in Application, g is a graphics object of your making
        GL11.glFinish(); // OpenGL is an object, GL11 is OpenGL 1.1, you can use whichever version you like
        Display.sync(60); // tries to stabilize your game at 60fps
    }
    Display.destroy();
    AL.destroy();
    System.exit(0);
}

// Display, DisplayMode, Sys, and AL are all objects given to you by LWJGL
Deferred Lighting

- Advanced rendering technique for pixel perfect lighting
- All lighting calculations are done in a shader
  - In 3D, this means recording the 3D screenspace vector of every pixel position on screen
  - In 2D, your game is orthographic, meaning that using the scale and translation of the viewport, we can easily convert from screen position to world coordinates
- Called deferred lighting because we draw a fully lit world, and then defer the lighting to a second drawing pass where we multiply each pixel by the light value (from 0-1) at that pixel
  - This second pass is done in a shader, which for our purposes is a small program that is applied to each pixel on screen and returns a color for that pixel
Deferred Lighting

- Overall strategy (simplified for 2D):
  - Draw the scene, storing the color in a texture.
    - Use an OpenGL FrameBuffer for this
  - Draw all the lights as circles
    - use a shader to combine the light value of the circle with the scene texture, thereby “revealing” the underlying scene
Deferred Lighting

• Things that the lighting shader will need to know about:
  – Scale and translation of the viewport
  – Radius, color, and position of the light being drawn
  – The texture that the world was rendered to
Particles

• What is a particle?
  – A particle is a tiny entity that is used in massive quantities to create a visually pleasing effect
  – Usually don’t affect the gameplay in any significant way
  – Commonly used for explosions
Particles

• What makes particles look good?
• Fade out or get smaller linearly over their lifespan
  – Once the particle is completely gone or transparent, it can be removed from the world
• Adding some kind of randomness to how they move
  – Starting position, velocity, acceleration, color, size, shape
Particles

• Particles are great
• But they are very slow if not done correctly
• Things that make them slow:
  – It’s a lot of information to tick
  – It’s a lot of information to draw
  – There are way too many of them to consider doing collision detection against each other
Particles

- **Optimizations? Get ready.**
- **Reduce the amount of information in your particles**
  - Vec2f position, Vec2f velocity, boolean alive
  - maybe some noise values to make them scatter
  - Less information to tick
- **Don’t make your particles Entities**
  - Keep them in a separate list so that you can tick and draw them all at once
  - If you are using OpenGL, then binding the particle texture once and then drawing all your particles without unbinding and re-binding the texture is a HUGE improvement
- **Don’t collide them with each other**
  - That’s a lot of math
  - If you must have your particles collide, have them collide only with entities, not with each other
  - This means they also don’t need a shape, so they take up less space
- **Keep them in an array**
  - This limits the number of particles you can have (which might be a good thing)
  - Keeps all the memory contiguous
  - Once you are trying to allocate more particles than you have room for, the oldest ones are kicked out first
- **Tick them in your draw loop**
  - But you said to never do that!
  - I know, sorry – but in this instance the speed gain of only having to iterate over them once is worth the terrible design
Parallax

• Used for foregrounds and backgrounds
• Each background is scaled by a little bit less, and then translated so that it is offset
  – Something that is scaled by .5 will translate half as much as something scaled by 1, making backgrounds that are farther back move slower
Lecture 10
Tips for Final 1
Plan Ahead

• Integrating engines can take time
• Implementing new engine features has to happen in ~2 weeks
• Make realistic expectations with your mentor TA
• Hold yourself to high standards – make something you’re proud of!
Tips for Final 1

JAVA TIP OF THE WEEK
Constructors are Lame

- Superconstructors put restrictions on the order complex calculations can be performed
- Leads to duplicate code in constructors
- We’d also like to have initialization near the variable source

```java
public class MyClass {
    private int i;
    private String str;

    public MyClass(int i) {
        this.i = i;
        str = "I’m number " + i;
    }

    public MyClass(OtherClass other) {
        // We can’t put code before this
        this(0);
        for (...) i += 1;
        str = "I’m number " + i;
    }
}
```
Initializer blocks!

• Unlabeled blocks of code directly in the class body
• Initializer blocks solve problems with duplicated constructor code and allow initialization to be performed at the variable declaration
• Executed from top to bottom when the class is instantiated

```java
class InitBlockExample {
    public static final String s;
    static {
        String temp;
        // complicated logic here
        s = temp;
    }
}
```
Field initialization shorthand

- Field initialization is just shorthand for initializer blocks

```java
public class MyClass {
    private static int i = 12;
    private String str = ""
}
```
Good uses

• Immutable final collections
  – Lists, maps, etc.
• Keeping complicated initialization code near field
• Debugging!

```java
class GoodUses {
  static final Map<String, String> m;
  static {
    Map<String, String> t = /*...*/;
    // lots of puts here
    m = Collections.immutableMap(t);
  }

  int complicatedInit;
  {
    // complicated init code
  }

  GoodUses(int ap) {}
  GoodUses(int ap, String s) {}
  GoodUses() {}
}
```
Other “Fun” Stuff

• When you specify a main class to run, the JVM:
  – Loads class via reflection
  – Calls main() via reflection

• Thus, static initializers are actually run before main()
  – Can System.exit(0) at the end of the static initializer to exit gracefully rather than crash with NoSuchMethodException

• Don’t ever do this

```java
public class Mainless {
    static {
        String s = "Look, ma! ";
        s += "No main!";
        System.out.println(s);
        System.exit(0);
    }
}
```
Tips for Final 1

QUESTIONS?
GAME DESIGN 8
Aesthetics
Remember MDA?
Extra Credits!

• Great video series on video games and game design
• Two episodes today:
  • Graphics vs. Aesthetics: https://youtu.be/5oK8UTRgvJU
  • Aesthetics of Play: https://youtu.be/uepAJ-rqJKA
What to Remember

• The MDA framework, that mechanics create dynamics which create aesthetics

• The designer (you) approaches from the mechanics side, but the player approaches from the aesthetics side

• Decide on your aesthetics early on, then work to bring them to life!
M3 Playtesting!
Last class playtesting!