Final TAs

- Jeff
  - Ben, Carmen, Audrey
- Tyler
  - Ruiqi
  - Mason
- Ben
  - Loudon
  - Jordan, Nick
- David
  - Trent
  - Vijay
Final Design Checks

• Optional this week
• Required for Final III
Playtesting Feedback

• Need three people to playtest your game
  – One will be in class
  – Two will be out of class (friends, SunLab, etc.)

• Should watch your victim volunteer play your game
  – Don’t comment unless necessary
  – Take notes
  – Pay special attention to when the player is lost and when the player is having the most fun
LECTURE 8
Networking
Networking

STRATEGIES
The Illusion

• All players are playing in real-time on the same machine
• 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 bullets
  - Fighter: timing is crucial
Modeling the World

• If we’re sending everything, we’re modeling the world as if everything is equally important
  – But it really isn’t!
  – Composed of entities, only some of which react to the player

• We need a better model to solve these problems
Client-Server Model

- One process is the authoritative server
  - Now we don’t have to wait for slow players, just the server
  - Server can either be hosted by a player or on a separate computer
- Other processes are “dumb terminals”
  - 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

```
Player 1 (server)
State 1
processP1Inputs()
processP2Inputs()
sendState()
State 2

Player 2 (client)
State 1
sendInputs()
receiveState()
State 2
```
Client-side Prediction

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

**Server**

- State 1
  - processP1Inputs()
  - processP2Inputs()
  - sendState()

- State 2

**Client**

- State 1
  - sendInputs()
  - processP2Inputs()
  - receiveState()

- State 2
Rollback

• But the server just sent a state that was 100ms in the past!

• 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 and with masking:
  - In an FPS, would need to lead shots because the game state displayed is in the past
- With rollback and without masking:
  - The target could be shot after they think they’ve taken cover (by a less laggy player i.e. the server player)
  - 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 Strategies

QUESTIONS?
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 (connecting to server, initial game state, 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 (e.g., current game state)
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 one-time messages
  - UDP streams updates
- Best choice varies by project
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;
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;
stream = new ObjectInputStream(in.getInputStream());
System.out.println(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!
Networking

QUESTIONS?
LECTURE 8
Procedural Generation
Procedural Generation

WHITE NOISE
What is noise?

- Randomness
- e.g. From 0 to 14 take a random number between 0 and 1
- By itself, it is jagged and not useful
White Noise

// returns a pseudorandom noise value for a given position
float noise(Vec2i vec) {
    Random r = new Random();
    r.setSeed(vec.hashCode());
    return r.nextFloat();
}
Procedural Generation

VALUE NOISE
Value Noise

• Smooth white noise by taking an average of neighbors
• Turns white noise into something useful
Value Noise

// returns a weighted average of the 9 points around the Vec2i v
float valueNoise(Vec2i vec){
    // four corners, each multiplied by 1/16
    corners = ( noise(vec.x-1, vec.y-1) + noise(vec.x+1, vec.y-1) +
                noise(vec.x-1, vec.y+1) + noise(vec.x+1, vec.y+1) ) / 16
    // four sides, each multiplied by 1/8
    sides = ( noise(vec.x-1, vec.y) + noise(vec.x+1, vec.y) +
              noise(vec.x, vec.y-1) + noise(vec.x, vec.y+1) ) / 8
    // center, multiplied by 1/4
    center = noise(vec.x, vec.y) / 4
    return center + sides + corners
}
Procedural Generation

INTERPOLATION
Interpolation

• Most interpolation functions take three arguments.
  • $a$ and $b$, the value to interpolate between.
  • $t$, a value between 0 and 1.
    – When $t$ is 0, function returns $a$
    – When $t$ is 1, function returns $b$
Interpolation

- Option 1: linear interpolation
- For values $a$ and $b$ and interpolation parameter $t$:
  $$f = a \times (1 - t) + b \times t$$
Interpolation

• Option 2: cosine interpolation
• \( t' = \frac{1 - \cos(t \times \pi)}{2} \)
• \( f = a \times (1 - t') + b \times t' \)
• Slower, but much smoother
Interpolation

- Option 3: cubic interpolation
- \[ t' = 3t^2 - 2t^3 \]
- \[ f = a \times (1 - t') + b \times t' \]
- Similar to cosine
Interpolation

• Option 4: Perlin interpolation
• \( t' = 6t^5 - 15t^4 - 10t^3 \)
• \( f = a \times (1 - t') + b \times t' \)
• Slightly slower than cubic
• Super smooth
Fractional Coordinates

• What if our x and y aren’t integers?
• Just find the values along the vertices of the unit square and interpolate
Fractional Coordinates

// returns the noise interpolated from the four nearest vertices
float interpolatedNoise(Vec2f vec){
    Vec2i topLeft = Vec2i((int) vec.x, (int) vec.y);
    Vec2i topRight = Vec2i((int) vec.x + 1, (int) vec.y);
    Vec2i botLeft = Vec2i((int) vec.x, (int) vec.y + 1);
    Vec2i botRight = Vec2i((int) vec.x + 1, (int) vec.y + 1);

    float dx = vec.x - ((int) vec.x);
    float dy = vec.y - ((int) vec.y);

    float topNoise = interpolate(valueNoise(topLeft), valueNoise(topRight), dx);
    float botNoise = interpolate(valueNoise(botLeft), valueNoise(botRight), dx);

    return interpolate(topNoise, botNoise, dy);
}
Named for its creator, this guy, Ken Perlin.

It’s a great way to make smooth, natural noise which can be used to create terrain, cloud patterns, wood grain, and more!

But you’ll probably use it for terrain...
Recall: Value Noise

• Smooth white noise by taking an average of neighbors

• Turns white noise into something useful
Perlin Noise

• Assign each vertex a pseudorandom gradient

Vec2f gradient(Vec2i vec) {
    float theta = noise(vec) * 6.2832;
    return new Vec2f(cos(theta), sin(theta));
}
Perlin Noise

- The noise value of each vertex is the dot product of its gradient and the vertex to the target point.
Perlin Noise

• Interpolate between the noise values of the four vertices (just like for value noise)
Perlin Noise vs Value Noise

- Value noise is easier
- Perlin noise has fewer plateaus
Procedural Generation

ADDING NOISE
### Adding Noise Functions

<table>
<thead>
<tr>
<th>Freq.</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amp.</td>
<td>1</td>
<td>$\frac{1}{2}$</td>
<td>$\frac{1}{4}$</td>
<td>$\frac{1}{8}$</td>
<td></td>
</tr>
<tr>
<td>Noise</td>
<td><img src="image1.png" alt="Noise_1" /></td>
<td><img src="image2.png" alt="Noise_2" /></td>
<td><img src="image4.png" alt="Noise_4" /></td>
<td><img src="image8.png" alt="Noise_8" /></td>
<td><img src="image_result.png" alt="Result" /></td>
</tr>
</tbody>
</table>

The table shows the addition of noise functions with different frequencies and amplitudes. The result is the combined noise function.
A Good Noise Function

• What does our noise function need?
  – Given an \((x,y)\) pair and a seed, returns the same value between 0 and 1 every time

• Random.setSeed() only takes a single seed as an argument
A Good Noise Function

• TA suggestion: use the Vec2f.hashcode() method
  – Returns a single integer that is unique to each pair
  – Will return the same integer every time

• Use this number to generate your seed for Random.nextFloat() call
// returns a pseudorandom noise value for a given position
float noise(Vec2i vec) {
    Random r = new Random();
    r.setSeed(vec.hashCode());
    return r.nextFloat();
}
• What follows is a lot of pseudocode that contains concepts that we haven’t discussed
  – Persistence, octaves, etc.
• Use this website as a reference for value noise:
  – Also covers an even smoother version of cubic interpolation
• Use this website as a reference for Perlin noise:
  – We stole our Perlin noise pictures from them
Procedural Generation

QUESTIONS?
M III / Tac V

Playtesting!!! :D