Lecture 7
Announcements
Level Editor

• Should be stable (as of Sunday)
• Will be updating the source code if you want to make tweaks to it for your final
Spooooooky

- Three checkpoints due next week
  - M II
  - Tac II
  - Final I
Spoookky

- Checkpoints split between two projects
  - Buff Wiz or Nin
  - Final 1
Spooooooky

- Playtesting requirements:
  - Just your engine
- Primary requirements:
  - Also just your engine
- Secondary requirements:
  - Two extras for Wiz or Nin
Nin2 Feedback

• Make sure gravity isn’t too strong, and your jumping isn’t too weak
• Give your game walls so that bouncing objects don’t fly off into the void
Final

• PDFs will be sent out by Wednesday
• Requirements are somewhat vague and subjective
  – Secondary requirements are just polished version of primary requirements
• Your plans may be subject to change
• We want the rubric to be flexible enough to handle that
  – If you want to change engine/game requirements that are listed on your pdf, you can do that, just check with us first!
Final

• Final I is due next week
• There will be a week-long break between Final IV and Final V for Thanksgiving (we’ll still have lecture, but nothing will be due)
• **Everything** is due by the 20th
  – All retries for all projects
  – All primary engine requirements
Final Groups

• You can share code with your group members
• Please be honest with your submissions
  – If you’re missing Nin1, don’t hand in your group member’s Nin1
  – This is against Brown’s academic policy
Special Topics

• We’ll be posting ~4 slide decks each week, even if we only go over two per lecture

• This is so you can look over the slides if they’re relevant to your final engine
Lecture 7

Hang in there!

Sound
Sound in Games

- In the real world, computers have sound
- Background music
- Sound effects
- Can be an important part of gameplay
  - Listening for footsteps
  - Dramatic music
Sound File Formats

• Many ways to encode and store sound
  – mp3
  – m4a
  – wav
Sampled Audio

- Usually recordings of live sounds
- Samples of sound wave at regular intervals
- Prevalent in modern games
Generated Audio

- **MIDI**
- File provides information on instruments and notes
  - Similar to sheet music
- Sound cards translate from instruments/notes to sound
- Can instruct computer to play something even if you can’t play it
- Used to be popular to save space, not as common now
## Compressed vs. Uncompressed

### Compressed Sound Files
- Lossy or Lossless?
  - Lossy remove “least important” parts of sound wave
  - Lossless just use smart compression on raw wave
- Smaller file size (esp. lossy)
- Lossy is *slightly* lower quality
- Slower to decode and play
- Often used for music

### Uncompressed Sound Files
- Record as much as possible of sound wave
- Much larger file size
- Faster to decode and play
- Often used for sound effects
Buffering

- Decompressing and decoding is slow
- Use a buffer
- Buffers will be implemented in whichever sound library you are using
Sound Effects
Positional Sound

• Manipulates left and right speaker volumes based on sound’s source relative to player
Doppler Shifts

• Sounds moving toward you sound higher
• Sounds moving away from you sound lower
• Sounds traveling past you “slide”
Echo

- A sound is made, and it bounces off an object at reduced volume
Reverberation

• A sound is made, and it lingers from repeatedly bouncing off objects
• Similar to echo
  – Lasts longer
  – More scrambled/overlapped
Sound

IMPLEMENTATION
Use a Library
javax.sound.sampled

- **AudioSystem**: Provides factory methods for loading audio sources
- **Clip**: Any audio that can be loaded prior to playback
- **Line**: Any source of streaming audio
- **DataLine**: An implementation of Line with helpful media functionality (start, stop, etc)
- Other classes for mixing, ports, and other utilities

```java
File file = new File("mysound.wav");
InputStream in =
    new BufferedInputStream(
        new FileInputStream(myFile)
    );
AudioInputStream stream = AudioSystem.getAudioInputStream(in);
Clip clip = AudioSystem.getClip();
clip.open(stream);
clip.start();
```
javax.sound.midi

- **MidiSystem**: The AudioSystem for MIDI files
- **Sequencer**: Plays MIDI sounds
- Other classes for manipulation of instruments, notes, and soundbanks
  - Much harder to manipulate samples

```java
Sequence song = MidiSystem.getSequence(new File("mysong.midi");
Sequencer midiPlayer = MidiSystem.getSequencer();
midiPlayer.open();
midiPlayer.setSequence(song);
midiPlayer.setLoopCount(0);
midiPlayer.start();
```
Alternatives?

• Some drawbacks of the built-in sound classes...
  – Imprecise control over exact playback start/stop positions
  – Almost impossible to manipulate or even examine samples in realtime
  – While Java offers pan and reverb, other libraries offer more varied effects
• But it’s very effective for simple background music and sfx!
OpenAL

• Cross-platform audio API
• Pros:
  – Built for positional sound (distance attenuation, Doppler shift, etc. all built in)
  – More fine-grain control available
• Cons:
  – More involved than the default Java classes
  – Poorly documented
Others

• Most other libraries are platform-specific or wrappers for OpenAL
QUESTIONS?
Lecture 7

Data Persistence

*It's possible that you might have a problem*
What to Save?

• Settings
  – User profile
  – Game settings

• Game state
  – Progress through the game
  – Maybe the state of the world or current level
Where to Save?

• Data should be saved somewhere that is always accessible by your program!
  – Oftentimes the user’s home directory can be used for this purpose

• Saving data to the current directory will not work, as your program can be run from anywhere!
Data Persistence

PERSISTENT CONFIGURATION
User Settings

- Player name
- Custom controls
- Other In-game preferences
- Considerations
  - Need to save per user
  - Should be able to export between game instances
Saving Game Settings

• Preferred resolution
• Graphics detail level
• Considerations
  – Need to save per installation of game
  – Should not go in cloud storage – machine-specific, can’t “sync”
Strategies

• We’ve already covered XML!
• Use it
User Interface

- Don’t save display settings automatically, revert graphics changes if no response
Data Persistence

SAVING GAME STATE
When to Save Game

• Only at checkpoints
  – Easier to implement
  – Potentially more frustrating for player
  – Ensure they’re frequent enough
When to Save Game

• Any time at save stations
  – Like checkpoints, but user can go back and resave
  – Better for nonlinear games
  – Need to save level state/progress, but not exact positions (save room usually empty)
When to Save Game

- Whenever user wants
  - Harder to implement, need a “snapshot” of current game state
  - Good for difficult games with frequent failure
  - Can still restrict when user can save (e.g. not during combat)
Automatic Saving

• A good idea if the player is responsible for saving
  – Just because saves are available doesn’t mean the player will use them
• Don’t set user too far back when they fail
• Depending on implementation, can simplify saved state (ie, only save when no enemies are around)
User Interface

• Save slots
  – Easy, simple, annoying
• Native file browser
  – Easy way to allow arbitrary saves
  – Doesn’t mesh well with game, unprofessional
User Interface

- Custom save-file browser
  - Harder to implement, but most flexible/featureful
- Features
  - Screenshot of saved game
  - Show only current player’s saves
  - Sort by time & type of save
Error Handling

• What if the save file is corrupted?
  – Don’t exit with a stack trace
QUESTIONS?

Data Persistence
LECTURE 7

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
// 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 $t$:
• $f = a \cdot (1 - t) + b \cdot t$
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);
}

Procedural Generation

PERLIN NOISE
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)
Procedural Generation

PERLIN NOISE VS 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>(1/2)</td>
<td>(1/4)</td>
<td>(1/8)</td>
<td></td>
</tr>
<tr>
<td>Noise</td>
<td><img src="image1.png" alt="Image" /></td>
<td><img src="image2.png" alt="Image" /></td>
<td><img src="image3.png" alt="Image" /></td>
<td><img src="image4.png" alt="Image" /></td>
<td><img src="image5.png" alt="Image" /></td>
</tr>
</tbody>
</table>
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 Vec2d.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?
Nin 2 Playtesting
Hooray!