Platformer1 Feedback

• Week 1 of 3 is done!

• Collision debugger is almost done
  – Two things left we recommend implementing in the debugger

• By the end of this week, you’ll completely move your code to your own project
Confusion with Collisions

- t-value is in \([0, 1]\), meaning that the world space ray should have a direction that is equal to \((\text{endPos} - \text{startPos})\)
  - Not normalized!
- Collision code should return a struct, minimally containing:
  - t-value in \([0,1]\)
  - Normal
  - Point of contact
- You may want to put “fancier” stuff in later
- About 2-sided triangles
Final comes out this week!

• Get to thinking of a game you might like to make
  – To voxel or not to voxel?
    • Borrow concepts from Minecraft like Chunks and procedural terrain, but leave out the voxels?
    • Other things?
  – Add some “advanced” features
• Start asking around to see who you might want to work with
• Specifics coming at the end of this lecture (:}
QUESTIONS?

Announcements
LECTURE 8
Wavefront OBJ Models
Wavefront OBJ Models

EXTERNAL MODELS
What is it?

- File format specifying a set of triangles

What do we want them for?
- Character models
- Object models
- Levels models
- Navigation meshes (more on this next week)

Basically, anything you wouldn’t want to represent using just code
OBJ format

• Universal 3D object format
• Stored as plain text
  – Vertices (v)
  – Texture coordinates per vertex (vt)
  – Vertex normals (vn)
  – Faces (f)
  – Many, many more things
OBJ Support Code

• We provide support code that handles the following:
  – Reading an OBJ and generating triangles from it
  – Writing OBJ files
  – Efficient internal representation of triangles:
    • Hard to use for collisions!
    • You should use your own triangle representation for your game purposes
OBJ Support Code

• Treat OBJS as any other resource, like textures
  – Only import once
  – Specific instance properties like scale, texture, or animation should go in a containing class instead of modifying the OBJ

• Use the Shape class to render OBJs!
Wavefront OBJ Models

QUESTIONS?
LECTURE 8
Collision Response II
INTEGRATING COLLISIONS

Collision Response II
Introduction

• Where we left off:
  – World is modeled as a set of triangles
  – Game Objects are modeled as ellipsoids
  – Sweep test: when moving the ellipsoid through a displacement from A to B, find the ellipsoid position that first intersects the world

• Now we need to respond to the collision
  – Bounce, slide, or a combination of both

• Remember, the goal is not physical accuracy!
  – Real people are not ellipsoids
  – We’re modeling the player’s intentions
Introduction

- Ellipse is trying to move by vector $\vec{v}$
- Collision between an ellipse and a triangle should return a $t$ value between 0 and 1
  - 0 means the ellipse did not move at all, 1 means the ellipse moved the full length of $\vec{v}$
- Test for collisions by:
  1. Test ellipse against every triangle
  2. Take the minimum $t$ value that is greater than 0
  3. Move ellipse by $\vec{v}t$
- But we now have “left over” velocity
Bouncing

- Remaining velocity can be split into 2 components: one parallel and one perpendicular to the contact plane
- Keep the parallel component the same
- Negate the perpendicular component
  - Elastic (preserves kinetic energy)
  - Inelastic (some energy is lost)

\[ V_{\text{before}} = V_{\text{parallel}} + V_{\text{perpendicular}} \]

\[ V_{\text{after}} = V_{\text{parallel}} - V_{\text{perpendicular}} \]
Sliding

• Split remaining velocity into 2 components: one parallel and one perpendicular to the contact plane
  – Keep the parallel component the same
  – Zero the perpendicular component

\[ V_{\text{parallel}} \]

\[ V_{\text{perpendicular}} \]

Before

After
Sliding for player movement

• Move player forward to contact point using time of collision
  – But that alone would leave the player stuck (we have remaining velocity)
    • As if the ground had infinite friction
    • Instead, slide the player
  – Project the remaining displacement onto the contact plane
    • Make sure to use the remaining displacement—otherwise players would speed up after each collision!
  – Move player again with another collision test
    • This is recursive, but 2 or 3 iterations should be enough
Sliding for player movement
Sliding for player movement
Sliding for player movement

Original displacement

Leftover displacement
Sliding for player movement

Displacement for next iteration
Sliding for player movement

The next iteration could collide with another surface
Sliding for player movement

Left-over displacement
Sliding for player movement

Displacement for next iteration
Computing the contact point

- **Interior**: Use ray-plane intersection point
- **Vertex**: Use vertex position
- **Edge**: $P = \text{ray-cylinder intersection point}$
  $P' = [P \text{ projected onto } CD]$  
  $P' = \text{the contact point}$
- **Remember to do this in sphere space, then convert back to ellipsoid space**
Computing the contact plane

- **Contact plane**: passes through contact point and is tangent to ellipsoid
  - Isn’t always the plane containing the triangle
- Computing the contact normal
  
  \[
  R = \text{ellipsoid radius vector } (r_x, r_y, r_z) \\
  E/R = \text{ellipsoid center in sphere space} \\
  P/R = \text{contact point in sphere space} \\
  \text{Normal in sphere space } = E/R - P/R \\
  = (E - P)/R
  \]
Contact plane in ellipsoid space

- We have the contact normal in sphere space
  - Now we need to convert it to ellipsoid space
- First attempt: Multiply by ellipsoid radius $R$
  - Works for positions
  - Doesn’t work for normals!

Sphere space  \[ \rightarrow \] Incorrect normal in ellipsoid space

Multiply by $R$
Contact plane in ellipsoid space

• Remember from cs123: always transform normals by inverse-transpose matrix
  – Results in dividing by $R$ instead of multiplying
  – So normal in ellipsoid space $= \frac{((E - P)/R)/R}{R}$
Problem: Floating point imprecision

• Due to finite precision, the ellipsoid is unlikely to be exactly on the plane
  – May be slightly above
  – May be slightly penetrating
• Once player overlaps with the world, collision detection in next frame will fail
• Player will fall through the floor
Problem: Floating point imprecision

- Solution:
  - Move the player away from the plane after every collision
  - Move by epsilon along the contact normal
- But this can cause other problems...
Problem: Acute angles

- When surfaces form an acute angle, moving the player by epsilon can push it into a face
  - And the player will fall through the floor
Problem: Acute angles

• Need to move player out of from several planes all at once

• There are both iterative and analytic solutions
  – We’ll talk about an analytic solution next
  – But you aren’t required to implement it
Analytic solution for acute angles

• Find all triangle interiors the player intersects
  – Project player center onto plane
  – Check if projected point is inside triangle bounds
  – You can ignore vertices and edges
• Create a set of the triangles’ planes
  – Adjacent, coplanar polygons contribute only one plane
• Solve for new position given old position and plane set
Analytic solution for acute angles

- We can get away with handling only the 2-plane and 3-plane cases
  - 4 or more won’t happen in a reasonable world
Plane-plane intersection

• Given planes as (normal, distance-from-origin) pairs:
  \[ A = (N_A, d_A), \quad B = (N_B, d_B) \]

• The line of intersection is \( C + Dt \) where:
  \[
  D = N_A \times N_B \\
  C = \frac{N_A(d_A - d_Bk) + N_B(d_B - d_Ak)}{1 - k^2} \\
  k = N_A \cdot N_B
  \]
Analytic solution for acute angles

- Take the set of planes overlapping the player
- Move each plane forward by sphere radius + epsilon
- Use planes’ intersection to find new player position
  - 2 planes: Project position onto line of planes’ intersection
  - 3 planes: Set position to point of planes’ intersection
Collision response: Recap

• We now have collision detection and analytic collision response for an ellipsoidal player
  — Yay!

• But wait, there’s more!
  — We can tweak our collision response to better handle certain scenarios (ramps, ladders, ice, …)
LECTURE 8
Player Movement
Player Movement

THE PLAYER’S INTENTIONS
Before we start...

• You do **not** have to implement all of these
• You are only required to implement **one** of your choosing
• That said, all of these will make your movement marginally better
  – Feel free to play around
Problem: Sliding while standing

• Currently, player will accelerate down slopes due to gravity
• Doesn’t happen in real life because of static friction
  – Force that resists relative motion of two surfaces in contact
• Solution: Simulate static friction using a hack
  – Split movement into horizontal/vertical components
  – Only do sliding for horizontal movement
  – Do 1 collision iteration for y-component (use this to determine if the player is on the ground)
• May want sliding on steep slopes
  – Could enable vertical sliding based on steepness of slope (the collision normal)
Problem: Bouncing down ramps

• Player bounces if vertical velocity is set to 0 on contact

• Solution: When on ground, set vertical velocity to a small downward value instead of 0
Problem: Deflection on ramps

- Contact normal pushes player up and to the side
Problem: Deflection on ramps

- Player’s intent is to move in a straight line
- Current algorithm projects endpoint of sweep onto the closest point on the plane
- Instead, we want to project the endpoint of the sweep to a point on the plane straight ahead of player
Problem: Deflection on ramps

• Solution: modify horizontal movement
  – Slide toward point on contact plane straight ahead, but using the distance that non-hacked collision response would produce
  – Given leftover displacement $V$ and contact normal $N$:
    
    New displacement direction $= V - \frac{N \cdot V}{N \cdot \hat{u}} \hat{u}$
    
    New displacement length $= \|V - (N \cdot V)N\|$

• No sliding for vertical walls ($N \cdot \hat{u} = 0$)
  – Can’t move any further while moving straight
  – To fix this, set new velocity $= V - (N \cdot V)N$ as before, but only for vertical walls
Surface types

- How do we add ladders, ice, sticky surfaces, or rotating platforms?
- Annotate faces as special surfaces
- Surface type is returned as part of collision detection
- Use surface type to adjust collision response
Frictionless surfaces

- On a half pipe, player should have same speed on enter and exit
- Must approximate curved surfaces with flat polygons when using polygonal collision detection
- Problem: Projecting the velocity when sliding loses energy
Frictionless surfaces

• **Solution:** Scale velocity so energy is conserved
  – Same velocity direction as before
  – New velocity length: $||V_{original}||$

• Don’t want to always redirect velocity
  – This hack is only correct for a polygonal approximation
  – We shouldn’t redirect when hitting a surface head-on
  – Only redirect when $N \cdot V > -0.5$
Case study: Hacks in real games

• Hack to fix player-world intersections
  – If player-world intersection is detected, try nudging slightly along all 3 axes to see if player is now free
  – Used in Quake 2 by id Software

• Simple method for climbing over steps and small objects
  – Before moving, lift player up a small amount
  – After moving, drop player down that same amount (unless player intersects an object)
  – Used in the game MDK2 by BioWare
Case study: Hacks in real games

- Easy collision response against a heightfield
  - `object.height = max(object.height, terrainHeight(object.horizPos))`
  - Used in many RTS engines

- Maximize floating-point precision
  - Floats have more precision near the origin
  - World is divided into sectors, each with its own local coordinate space
  - When player moves between sectors, objects positions are updated to the new origin
  - Used in Dungeon Siege by Gas Powered Games
Conclusion

• Collision response is a pile of hacks
  – Optimal player movement is not physically correct
  – Floating point imprecision is tricky
  – What we presented is definitely not the only way to do it
LECTURE 8
Tips for Platformer 2
Support Code!

- `<obj.h>` and `<obj.cpp>` do all things OBJ-related!
  - In `/course/cs195u/asgn/platformer`
- Important parts of `obj.h` / `cpp`
  - `OBJ::triangles` — the triangles of the OBJ, you’ll probably convert these to your own triangles, and collide your game objects with them
  - `OBJ::vboData` — vertex data for the VBO. You can create a shape from this using `std::make_shared<Shape>(vboData)`
- Feel free to extend or modify these classes to your heart’s content
Level files!

• 3 obj maps provided:
  – Easy: closed world, very few acute angles, great test area in a side room
  – Hard: closed world, some acute angles, also has a large test area
  – Island: open world, half-pipe approximation, hidden rooms, and more!

• Textures provided!
  – Janky baked-in lighting and shadows!

• All should allow a reasonable size entity to spawn at the origin
Collision Code Tips

• Make your “realism” hack toggleable so you can check to see it’s actually making a difference

• Watch out for division-by-zero!
  – Relatedly, watch out for normalizing 0-length vectors
LECTURE 8
C++ Tip of the Week
C++ Tip of the Week

FRIENDSHIP
Friend classes

- By default, only public variables and functions in a class are externally visible
- Public and protected variables/functions are visible to subclasses
- A *friend* class gets access to ALL members of a class, regardless of listed scope
Friendship

• Friendship is NOT symmetric by default

• It’s also not transitive:
  – If class C says class B is a friend, and B says A is a friend, C does not become a friend of A

• Subclasses don’t really inherit friendship
  – If B declares A as a friend, SubA does not inherit that friendship
  – SubB’s inherited members from B will still be visible to A
Making friends

class TA {
    private:
        gradeStudentGame(Student *s); // Should probably look at the source code
}

class Student {
    friend class TA; // Grants TA access to all Student’s members
    public:
        Game* playtestGame();
    private:
        QFile* accessSourceCode (); // This is now accessible to the TAs!
}
Friend methods

- You can also declare a single method as a friend:

  ```cpp
class TA {
    private:
      gradeStudentGame(Student *s); // Should probably look at the source code
      accidentallyRmFileSystem();  // Shouldn’t have access to student’s code
  }

class Student {
    public:
      Game* playtestGame();
      friend TA::gradeStudentGame(Student *s); // Grants access only within this function
    private:
      QFile* accessSourceCode();
  }
```
Operator Overloading

• Friendship is useful for making overloaded operators work
  – Since operators aren’t (usually) member functions of the classes they are being overloaded for, they won’t have access to private members of those classes
  – Solution: declare those operator functions as friends in your class
One more note - Functors

• Mysterious C++ ‘function objects’
  – Just classes with an overloaded () operator
  – Example: pass a comparator object into a priority queue to sort objects according to comparator()
  – Thanks to mjm9
LECTURE 8
C++ Anti-Tip of the Week
Some fun stuff

// What does this do?
!ErrorHappened() ??!??! HandleError();
// The same thing as this:
if (ErrorHappened())
  HandleError();

int c = 0;
// I love comments yeah wooooo?????????????????????????
c++;
// What does this print?
printf("%d\n", c);
Preprocessor trigraphs!

- `??` is actually an invalid character sequence in C
  - Literally not in the dictionary that defines the language
- Gets pre-processed into some other character based on whatever comes after it
- Great way to obfuscate code even more!

<table>
<thead>
<tr>
<th>Trigraph</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>??=</code></td>
<td>#</td>
</tr>
<tr>
<td><code>??/</code></td>
<td>\</td>
</tr>
<tr>
<td><code>??'</code></td>
<td>^</td>
</tr>
<tr>
<td><code>??(</code></td>
<td>[</td>
</tr>
<tr>
<td><code>??)</code></td>
<td>]</td>
</tr>
<tr>
<td><code>??!</code></td>
<td>!</td>
</tr>
<tr>
<td><code>??&lt;</code></td>
<td>{</td>
</tr>
<tr>
<td><code>??&gt;</code></td>
<td>}</td>
</tr>
<tr>
<td><code>??-</code></td>
<td>~</td>
</tr>
</tbody>
</table>
Why would I ever do this?

- You wouldn’t!
  - In fact, you have to specifically enable them to work
- Apparently IBM *really* wanted to keep them in the language spec
  - Deprecated for C++0x
  - Removed from C++17 and forward
C++ Tip of the Week

QUESTIONS?
LECTURE 8

Final Project Overview
Overview

• Can be any 3D game
• You should work in groups!
• Each person is responsible for 5 “points” worth of new engine features
  – More members in a group means more engine features
  – More details in the final project handout
  – If you are doing the class for a capstone, you will need 7 “points”
  – Must do at least one 3 “point” feature each
Timeline

• 4 main parts:
  – Week 1: Idea
  – Week 2: Form groups and get approved
  – Week 3: Design
  – Weeks 4-8: Code, playtest, polish, present
Week 1 (This Week): Idea

• A 1 page document
• Describe basic gameplay idea
  – How is your game fun?
  – Why should someone want to help make it?
• Describe engine feature(s) you plan on implementing
• Give a 90-second “elevator pitch” of your game in class
• Everyone must give a pitch, even if you already know your group and which project you’re working on
Week 2 (After Spring Break): Groups

• Form a group (or decide to work alone) – 2-3 people ideally
• Finalize game and engine features
• Each group must meet with a TA to present the following:
  – A more polished idea of the game
  – Breakdown of member responsibilities for engine
Week 2 (After Spring Break): Design

• Research new engine features
• Design the engine and game
• Exact breakdown of member responsibilities
• Choose someone’s engine to use or integrate engines
• For groups - explain how you will use version control
Weeks 3-8

• Week 3/4:
  – Engine should be mostly done
  – Game exists

• Week 5:
  – Engine should be done
  – Game is playable
  – 5 playtests per member from people not in CS1972
Weeks 3-8

• Week 6:
  – Game should be mostly done
  – 5 more playtests per member from outsiders

• Week 7:
  – Game should be done
  – 5 playtests per member
  – Powerpoint slideshow for postmortem presentation
Weeks 3-8

- Week 8:
  - Polish up your game, bug fixes, etc
  - Create an executable and put it in /contrib
  - Make a video demo of your game from gameplay footage

- It is now the end of the semester

- And then you’re done!
LECTURE 8
No Playtesting
Sign up for Design Checks!