LECTURE 5
Announcements
Falling Behind?

• Talk to us
  – You still pass the class if you hand in all projects at the end of the semester

• Hand in even if you think you won’t satisfy the playtesting requirements
  – You’ll get valuable feedback from the TAs and your peers
  – No advantage of not handing in
Camera Lag

• Right now
  1. Position of camera set to player position (tick)
  2. Player position updated (tick)
  3. Screen rendered (draw)

• Because of this, the camera lags slightly behind the player if the player moves

• Solution
  1. Player position updated (tick)
  2. Position of camera set to player position (afterTick – new event triggered by the world once all systems have ticked)
  3. Screen rendered (draw)
Graphics

• Lighting looks a little bland
• Can use built in Light class (graphics/Light.cpp) as well as Graphics functions add/setLight
  – Supports point lights, ambient lights, directional lights
• By default, maximum one light per scene
QUESTIONS?
LECTURE 5
Raycasting
What is it?

• Given a ray what is the first thing it hits?
  – Like shooting an infinite laser
• Can be used in graphics to find what to draw at each pixel
• Can be used in games for tons of things
  – Line-of-sight AI
  – Hit-scan weapons
The Ray

- A ray is a point (source) and a direction
- A point on ray \( \vec{r} \) is given by:
  - \( \vec{r} = \vec{p} + t\hat{d} \)
- \( \vec{p} \) is the source point
- \( \hat{d} \) is the direction
  - This must be normalized!
- \( t \) is a scalar value (length)
Why Raycast?

- Alternative: move a small “trajectory” object through the world very quickly

- Few large steps
  Collision detection is piecewise

- Many small steps
  Too expensive
Why do we care?

• Best answer to a very important question…
• What is the camera looking at?
• In minecraft, this probably determines:
  – Which block am I trying to destroy?
  – Where am I trying to place a block?
  – If I shoot a weapon, what does it hit?
IMPLEMENTATION
Raycasting in Minecraft

• Cast a ray from the camera eye in the direction of the look vector.
• The first intersection with the environment gives you:
  – The block the player is currently looking at
  – The face of that block the player is looking at
• The catch: we do this every tick
  – So it needs to be super-fast
  – Can’t just intersect every visible block face
Voxel Ray Traversal

- Take advantage of the fact that we’re traversing a grid!
- Break ray into components based on grid boundaries
- Compute the $t$ value (distance) required to move the ray to the next component
- Continue checking grid boundaries until intersection with a solid block
Voxel Traversal Algorithm

// +1 or -1 depending on direction
const int stepX, stepY;
// t delta to span an entire cell
const float tDeltaX, tDeltaY;

// integer block coordinates
int X, Y;
// t value to next integer boundary
float tMaxX, tMaxY;

loop {
  processVoxel(X, Y);
  if (tMaxX < tMaxY) {
    tMaxX += tDeltaX;
    X += stepX;
  } else {
    tMaxY += tDeltaY;
    Y += stepY;
  }
}
Voxel Traversal Algorithm

- **Be very careful initializing step, tMax, and tDelta**
  - Sign & rounding issues will ruin your day
- **Be sure to normalize this vector in your own code!**
  - When computing tDeltaX, tDeltaY, tDeltaZ
- **Be careful of rays parallel to an axis (divide by 0)**
- **Look at the paper for more details**
  - [http://www.cse.yorku.ca/~amana/research/grid.pdf](http://www.cse.yorku.ca/~amana/research/grid.pdf)
Tips

• Add a visualization – intersected block, face, and point
  – This is actually required for Minecraft3
• Get *one* direction working first
  – For example, rays, in the (+x, +y, +z) direction
• Test *all* directions afterwards
  – Shouldn’t have to special case the loop!
QUESTIONS?
LECTURE 5
Dynamic World Loading
Dynamic World Loading

MEMORY MANAGEMENT
Memory troubles

- Assuming a block is one byte, one chunk is $32 \times 32 \times 32 = \sim 33$ KB
- Last week we had 200 chunks = $\sim 6.3$ MB
  - Decent size world, but not huge
- We want our worlds to be “infinite”
  - So big that reaching the end is unreasonable during standard play
- But we don’t have infinite memory...

Your RAM: 8GB
Assassin’s Creed Unity: 50GB
What should we forget?

• Solution: store only in memory what the player is likely to interact with

• Two parts:
  – Load chunks only as the player approaches them
  – Unload chunks when the player has moved significantly far away from them
Dynamic World Loading

CHUNK STREAMING
Chunk streaming

- Only store chunks within a distance from the player
  - AABB or sphere around player
- Update when player moves between chunks
  - Remove chunks that went out of range
  - Add chunks that came into range
• What if the player transitions from chunk \((x, y, z)\) to chunk \((x+1, y+1, z+1)\)
  – If view distance is 5 chunks on each side of the player, need to stream in more than 50 chunks
  – Too much work for a single frame

• Simplest solution: queue of added chunks
  – Dequeue one chunk per frame
  – Build chunk’s Shape when it’s dequeued
More Complicated Solutions

• Use several worker threads
  – Cannot access OpenGL in other threads
  – Shape initializations still need to be spread out across frames
  – Many functions aren’t thread-safe (like srand() and frand())

• Don’t purge chunks in the world until they are (1+radius) away instead of (radius) away
  – Loading chunks into memory is hard, so keep them there for a little while longer
  – Keeps you from having to reload chunks when players jump back and forth across a chunk border (“thrashing”)
    • We prefer “flailing”
Saving and loading

- With chunk streaming, modifications to chunks are lost when it goes out of the player’s view range
- Could try to save all modifications in memory
  - Danger of running out of memory for very long play sessions
  - Doesn’t provide persistence across play sessions
- Solution: Save chunks to disk as they stream out, load them from disk as they stream in
- How to efficiently save and load so much data?
Saving and loading in Minecraft

• Only saves visited chunks
  – Pre-computing and storing entire world would be too much
    • Not as bad in 2D though (Terraria does this)

• Uses a system of files
  – Each 32x32 group of chunks is put in a “region” file
  – Region files are named “r.[x].[z].mcr”, where x and z are the region indices
  – 8KiB header with info in which chunks are present, when they were updated, and where they can be found

• Chunk data compressed with Zlib

• Not required!

• More information: http://www.minecraftwiki.net/wiki/Beta_Level_Format
Dynamic World Loading - Chunk Streaming

QUESTIONS?
LECTURE 5
Tips for Minecraft 3
Adding/removing blocks

• Significant part of gameplay!
• Adding blocks:
  – Raycast to determine intersected face
  – Add a block in the direction of the normal
• Removing blocks:
  – Directly remove the intersected block
• Debug visualizations are your friend!
  – Actually required – visualize the intersect block and face
Adding/removing blocks

• Logically changing a block is easy
  – Change that block’s ID
  – Tell neighboring blocks to render faces if changed to a transparent block

• Visually changing a block isn’t
  – Have to re-build the entire Shape
  – Happening at most once per frame, so it’s okay
Edge cases

• What if...
  – The block you intersect is on a chunk boundary
    • Your algorithm shouldn’t care!
  – You try to place a block inside an entity
  – You try to place a block inside yourself

• You should try to handle these as well as possible
Interacting with Entities

• Gameplay requirement this week – there must be some entities to interact with!
• Our collision engine supports AABB-Chunk collisions and cylinder-cylinder collisions
Entities and You

• Entities should affect you!
• Use cylinder-cylinder collisions
  – Cylinder collision system
You and Entities

• You may want to be able to fight back!
• Do a distance check
  – Decent for making throwable “bombs”
• Use cylinder-cylinder collisions
  – Have cylinder “bullets”
• Use raycasting!
  – Ray-AABB is just intersecting 6 planes
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

```cpp
pos += 10;  // jerky movement

vel = 10;   // constant velocity
pos += vel * seconds;

acc = 1;
vel += acc * seconds;
pos += vel * seconds;  // smoother, but not perfect
```
Goal velocity

- **goalVelocity** set directly from arrow keys
- Gradually set **velocity** to **goalVelocity**
- By applying an acceleration
  
  \[ A = k(v_{goal} - v_{current}) \]
LECTURE 5

C++ Tip of the Week
The preprocessor

• Happens before the compiler
• Basically string replacement
• Good uses
  – #include
  – Header Guards
• Bad uses...

```c
#ifndef APPLICATION_H
#define APPLICATION_H

Class Application
{
  
  ...
}

#endif
```
#defines for constants

- A common use of defines is for constants, but this has problems
  - The compiler never sees it—if you get an error, it may show the value not the symbol (32 not CHUNK_SIZE)
  - It can’t be scoped
- Prefer const variables!
  - Seen by the compiler
  - Only ever one instance
  - Can be limited to class

```cpp
//bad
define CHUNK_SIZE 32

//good
const int ChunkSize = 32;

//scoped
class Chunk
{
  private:
    static const int ChunkSize = 32;
};
```
#define for functions

• Another common (mis)use of defines is for quick function
  – They reduce the number of function calls. So faster
  – Lead to problems due to string replacement
    • Parenthesis needed
    • Bizarre behavior
    • Scope

• Prefer inline functions!
  – Scoped
  – Type safety
  – Predictable
  – Just as fast

#define CALL_WITH_MAX(a,b) f((a) > (b) ? (a) : (b))

```cpp
int main()
{
  int a=5;b=0;
  CALL_WITH_MAX(a++,b);
  cout << a << endl;
}
```

```cpp
int main()
{
  int a=5;b=10;
  CALL_WITH_MAX(a++,b);
  cout << a << endl;
}
```

```cpp
inline void CallWithMax(float a, float b) //can use templates to make
{
  f(a>b ? A:b);
}
```