
cs195u: 3D Game
Engine Development

Welcome

- 3D game engine design for PC games
  - Not game design
  - You develop multiple engines from scratch
  - This is an intense but rewarding course
- Three individual projects
  - Weekly checkpoints
- Final group project
- Prerequisites: cs32, cs123, recommend cs195n
- Technologies: C++, Qt, OpenGL

Focus for this course

- Building a 3D game from scratch
- Using OpenGL to build 3D environments
- Player movement & interaction
- Collision detection & response

Does **NOT** focus on:

- Latest 3D rendering techniques
- Using existing 3D game engines (Unity, UDK, XNA, etc.)
- Realistic physics (rigid body, ragdoll, etc.)
- Server-side programming (MMO back-ends)
- *Advanced game AI
- *Game design

What is a 3D Game Engine?

- No such thing as a game engine that can support any game
- Core set of components to facilitate game creation
  - Rendering
  - Physics
  - Collision detection
  - Sound
  - Artificial intelligence
  - Real-time networking
  - Resource management

Real-Time Strategy (RTS) Engine

- Large number of low-detail game units
- Client/server networking with some lag tolerance
  - Lockstep protocol
- Heightmap-based terrain
- Scripting support for extensibility / modding

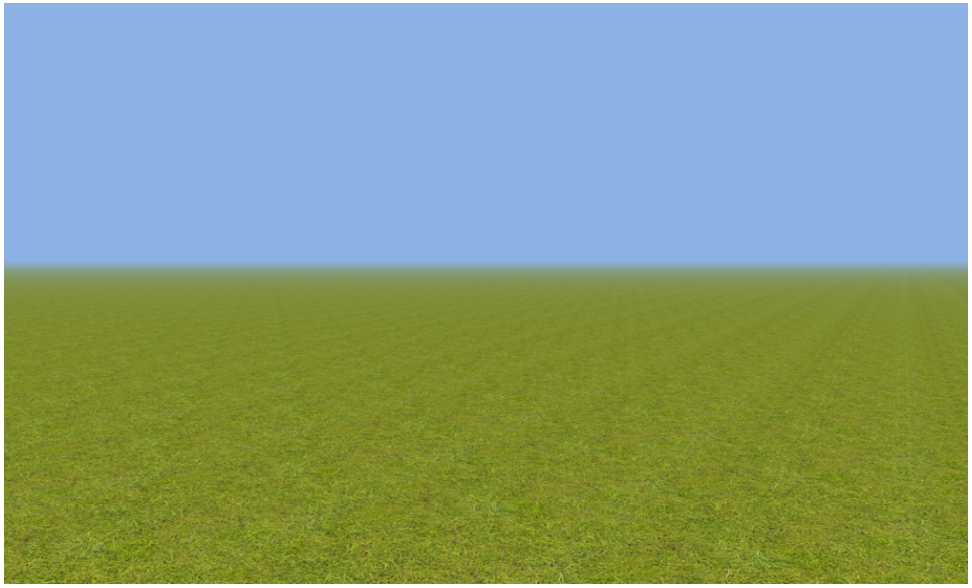
Racing Game Engine

- Low number of high-detail models
- Advanced realistic physics
- Peer-to-peer networking
  - Minimize latency between players
- Advanced graphical effects
  - Motion blur

Projects

Warm-up

- One-week starter assignment
- First-person movement
- OpenGL texture loading
- Practice using the support code



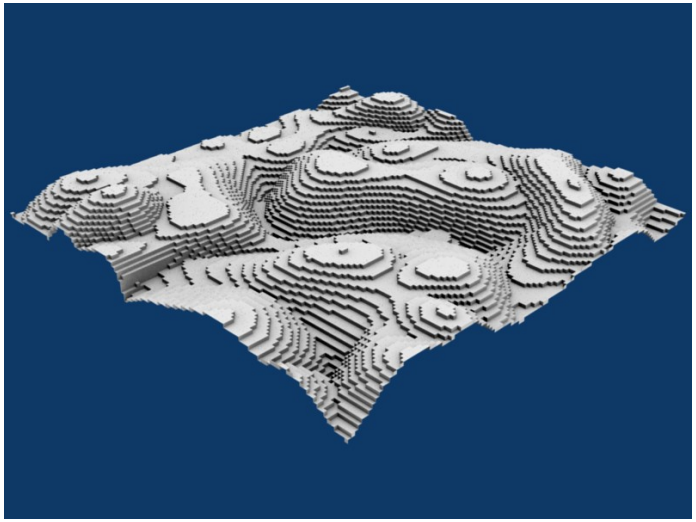
Demo

Minecraft: The Game

- Indie game that made millions
- Procedurally generated block world
- Simple pixelated graphics
- Undirected multiplayer gameplay
- Players love shaping their own world

Minecraft: Class Project

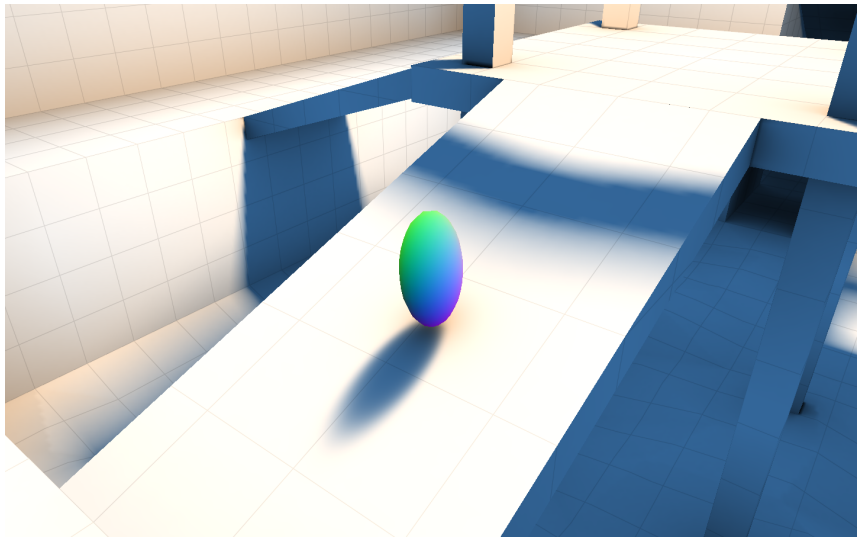
- Procedural noise for terrain generation
- Simple collision detection and response
- OpenGL rendering and animation
- View frustum culling
- Dynamic data set



Demo

Platformer

- Ellipsoid-mesh collision detection and response
- Advanced level representation
  - Arbitrary 3D levels
  - Lots of geometry and math



Platformer

- Path finding with a navigation mesh
- In-game UI
- More open-ended gameplay

Demo

Final Project

- Group project with up to five people
- Your own idea, developed and implemented
- Advanced engine features required, some ideas:
  - Gravity volumes
  - Networking
  - Scripting
  - Portals
- Playtesting with the general public
  - Polish will be important!
- You own your creation

Checkpoints

- Although there are only four projects, there are weekly checkpoints
  - Checkpoints will be due at noon on Friday
- Each checkpoint has well-defined objectives
  - Meet all objectives, get credit for checkpoint
  - Miss an objective, you have one week from the grading date to fix it

Grading

- Checkpoint requirements are minimal
  - You get out what you put in
- Assuming all playtesting completed:
  - A: all N checkpoints
  - B: ≥ N - 3 checkpoints
  - NC: < N - 3 checkpoints

Late Policy

- 2 late passes
  - Allows handin up to a week late
  - Whether you hand in 1 hour late or 6 days late, you still use 1 pass
  - You cannot use both late passes on the same assignment
- Don't fall behind!

Weeklies & Playtesting

- Weekly: 2-3 minute presentations by each student
- Playtesting: Playing other student's game and filling out questionnaires
  - Opportunity to give each other feedback
- Both will occur during class time
- Both are **required**
  - Missed weeklies / playtesting must be made up
- Dates for both are on the course calendar
  - Weeklies for intermediate results, playtesting for complete games

Collaboration Policy

Overview of Warm-up

- Get started with Qt Creator
- Minimal starter code
- Simple OpenGL calls
- First steps with game loop
- First-person movement
  - Move the camera with the mouse
  - Move the player with the keyboard

Starter Code

- Qt framework
  - Cross platform
  - Huge framework
  - You will need at least QGLWidget, QTimer, and QImage
- Qt handles the application loop
  - Your code will reside in callbacks
  - `QGLWidget::mousePressEvent(QMouseEvent *)`
- Suggest separating engine from QGLWidget
  - Cleaner and easier to port

Game Loop

- Separate logical steps, not necessarily in sync:
  - Handle user events
  - Read/write network data
  - Update game state
  - Draw game state
- Variable vs fixed timestep
  - Recommend `float seconds`, not `int milliseconds`
- Most games max out CPU, spinning as fast as possible
  - Starter code uses a timer, set timeout to 0 to do this
  - Don't do this if not needed (laptop battery life)

Horizontal Player Movement

- Simple trigonometry for horizontal movement
  - Player is rotated around the y-axis (up) at some angle
  - Just use sin and cos to get x and z velocity
  - `p.x += v.x * t`
  - `p.z += v.z * t`
  - `t` is the number of seconds elapsed since the last frame
- Strafing (side-to-side movement)

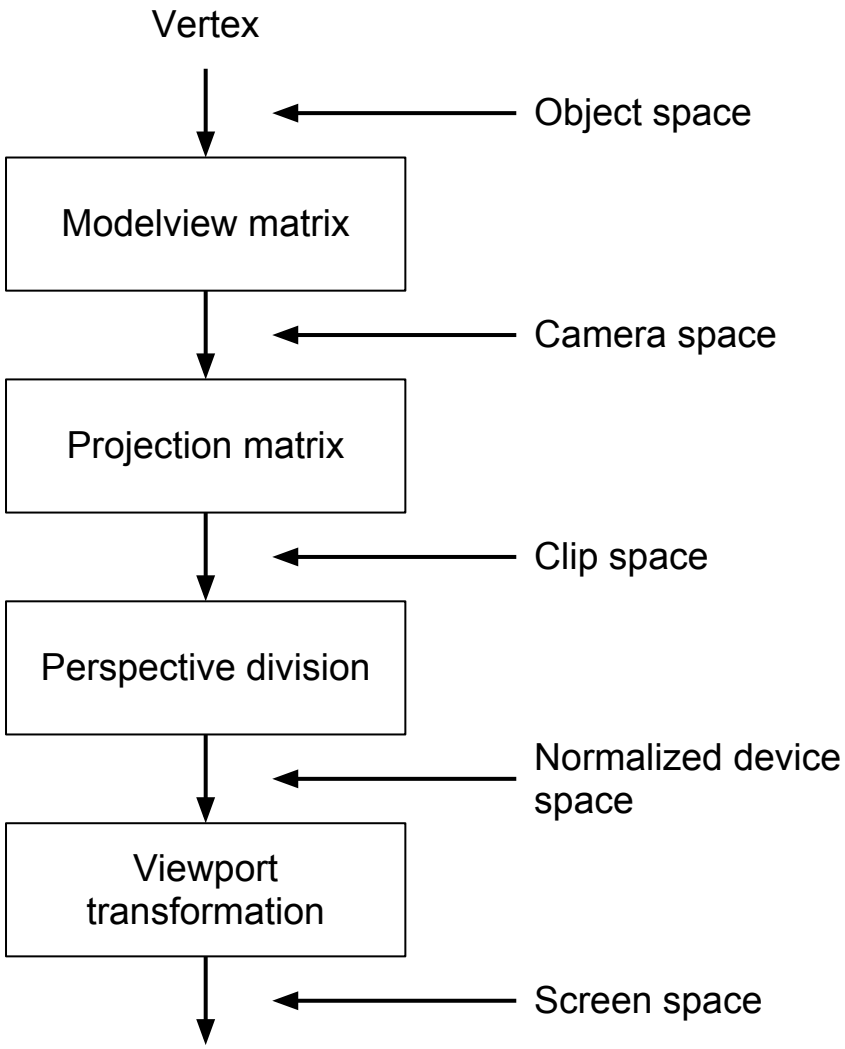
Vertical Player Movement

- Acceleration due to gravity
  - `p.y += v.y*t + 0.5*a.y*t^2`
  - `v.y += a.y*t`
- Collision with ground
  - `p.y = max(0, p.y)`
  - We will do collision differently later

OpenGL Matrix Transformations

- Ignore five matrices from CS123
  - OpenGL transformations use two matrices
  - Stored in column-major order (need to transpose from a C array)
- Projection matrix
  - Has camera parameters (field of view, aspect ratio)
  - Usually only changes on window resize
- Modelview matrix
  - Has transformation from object space to camera space
  - Usually changes at least once per frame

OpenGL Matrix Transformations



First-Person Camera

- Before drawing
  - Reset to the identity matrix
  - Rotate around X and Y axes
  - Translate to negative player eyepoint
- On window resize
  - Use `gluPerspective()` to set up camera parameters for the projection matrix
  - Don't forget to use `glMatrixMode()` to switch between modelview and projection matrices

Loading a Texture

- Simplest way uses QImage from the Qt framework:

```
QImage img;
img.load(path);
img = QGLWidget::convertToGLFormat(img);

unsigned int id;
glGenTextures(1, &id);
glBindTexture(GL_TEXTURE_2D, id);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, img.width(), img.height(),
    0, GL_RGBA, GL_UNSIGNED_BYTE, img.bits());
```

Drawing Geometry

- Easiest way uses OpenGL immediate mode:

```
glBindTexture(GL_TEXTURE_2D, id);
glEnable(GL_TEXTURE_2D);
glBegin(GL_QUADS);
glTexCoord2f(0, 0); glVertex3f(0, 0, 0);
glTexCoord2f(1, 0); glVertex3f(1, 0, 0);
glTexCoord2f(1, 1); glVertex3f(1, 1, 0);
glTexCoord2f(0, 1); glVertex3f(0, 1, 0);
glEnd();
glDisable(GL_TEXTURE_2D);
```

C++ Tip of the Week

- Helpful Qt things:
  - class QList<T>

    Like std::vector<T> but much easier to use and usually faster. Magically uses an array of T* internally when sizeof(T) > sizeof(T*). Also overrides += and << operators for quick manipulations.
  - class QHash<K, V>

    A hashtable that will be useful in this course. Override the global qHash() function to return the integer hash for new key types.
  - class QPair<A, B>

    A tuple of two types, this comes in handy for hash table keys.

C++ Tip of the Week

- Helpful Qt things:
  - foreach loop

    A macro that makes iterating over collections a lot easier. Requires the <QtGlobal> include:

```
QList<int> ints;
ints << 1 << 2 << 3;
foreach (int i, ints) {
    cout << i << endl; // Value is a copy
}
foreach (const int &i, ints) {
    cout << i << endl; // Value is a reference
}
```

Good Luck!

Warm-up is due next week