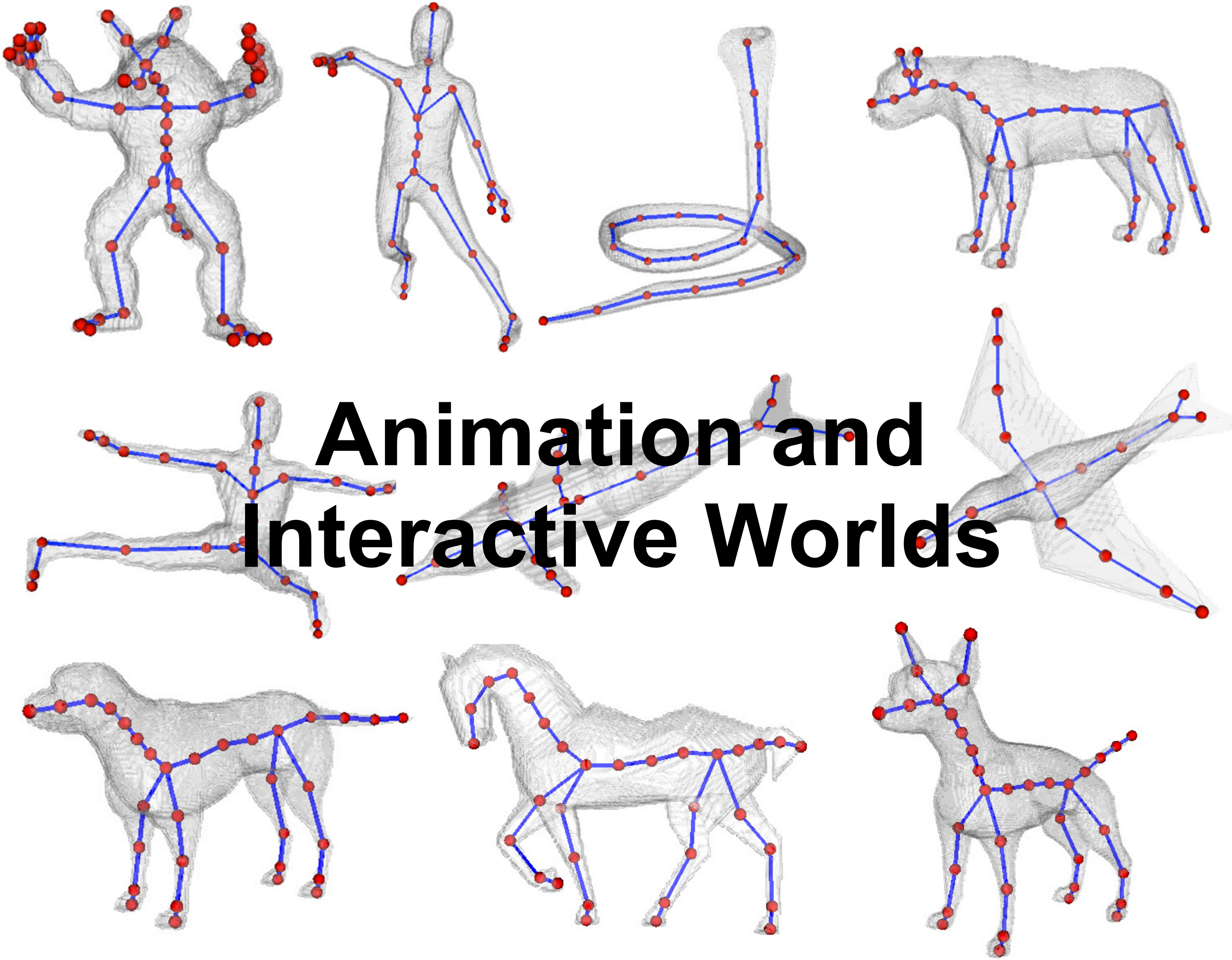
Animation and Interactive Worlds



Overview

Today we're covering several loosely-related techniques:

- Object representation
- Animation basics
- Minecraft week 3
  - Adding and removing blocks
  - Voxel traversal for ray-tracing
  - Simplistic enemies
  - AABB-AABB collision detection
  - Streaming chunks

Game Object Representation

- Many possible ways to represent a game object
- Usually want several representations
  - Different constraints for rendering, collision detection, physics, animation, etc.
- Game object as a collection of "components" used by different engine subsystems
  - Component-based engines
    - More on these in later lectures

Object Representation: Rendering

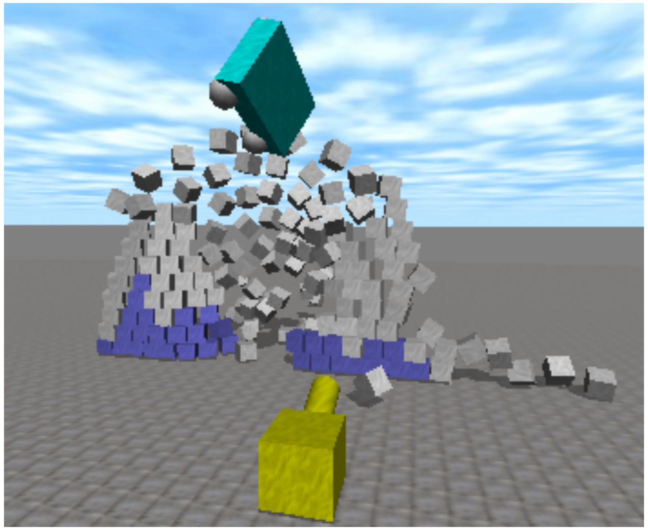
- Designed for efficient resource usage
  - Ordered by material
  - Geometry in formats meant for graphics hardware (e.g. triangle strips)
- Many objects are only for rendering
  - Blades of grass
  - Window sill on wall

Object Representation: Collision

- The simpler, the better
  - As long as the player doesn't notice an inconsistency with the visual result
- Common geometry for collision detection
  - Sphere / Ellipsoid
  - AABB / OBB
  - Cylinder
  - Capsule
- Some geometry is just for collision detection
  - Blocking volume on edge of cliff
- Detailed math next lecture!

Object Representation: Physics

- Simple geometry with extra properties
  - Center of mass & mass distribution function
  - Linear velocity & angular velocity
  - Coefficients of friction
- Joints enforcing collision constraints
  - Joint types include ball-and-socket, hinge, piston



Object Representation: Animation

- Baked animation
  - Interpolation between keyframes in a sequence
  - Full-character or per part (different leg and torso animations)
- Procedural animation
  - Generate pose dynamically from environment
  - Example: Create pose from contact points
- Attachment points for sword, shield, etc.

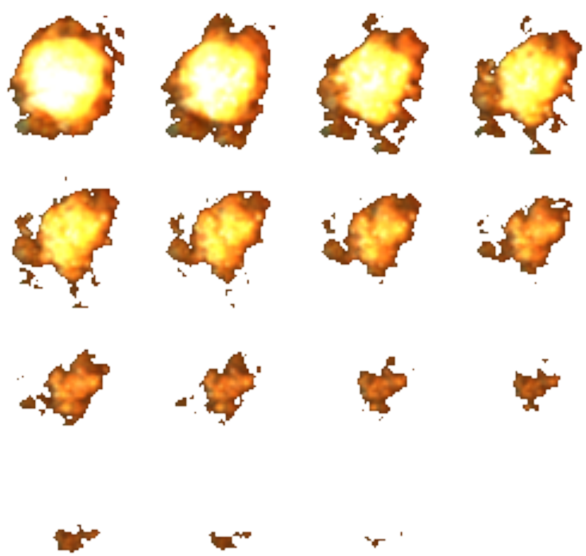
Animation

Main categories:

1. Sprite / texture animation
2. Vertex animation & morph targets
3. Skeletal animation

1. Sprite Animation

- Set of still images played in sequence
- Primarily used in 2D games
  - All entities in early 3D games like Doom
  - Particle effects in modern games (billboarding)



2. Vertex Animation & Morph Targets

- Motion data for each vertex in a mesh
  - Morph targets blend between extreme poses
  - Gives fine-grained control, but is very time-consuming
- Can be used with skeletal animation
  - Commonly used for facial expressions

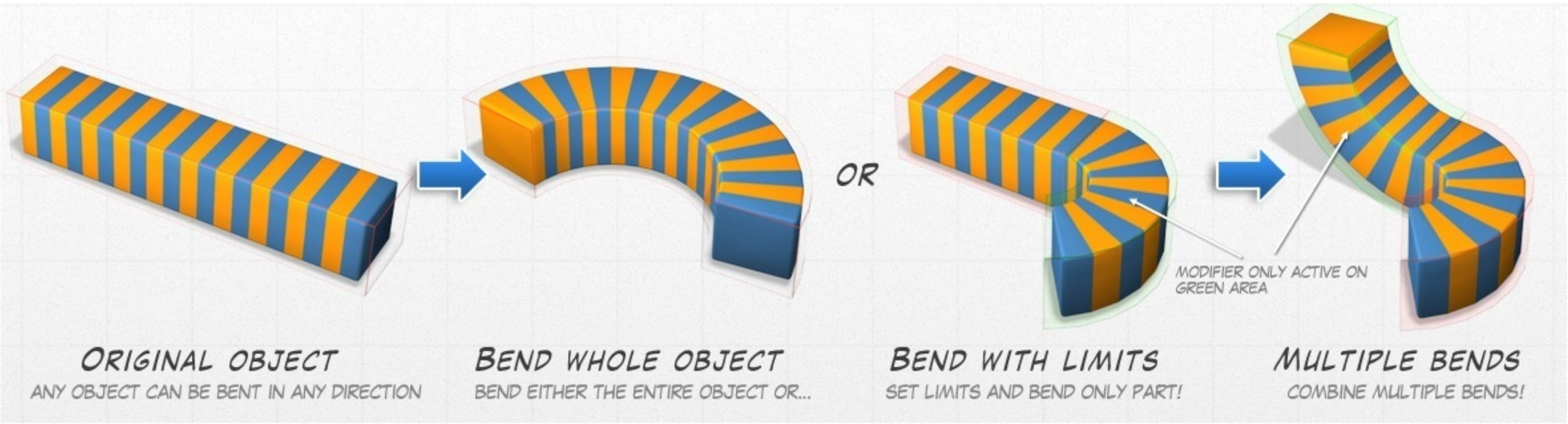
http://www.youtube.com/watch?v=_ebwvrrBfUc

Case Study: MD2

- Vertex-based format
  - Used in Quake II, by id Software (1997)
  - Saves all vertex positions for each keyframe
  - Baked necessary OpenGL commands into the file: GL_TRIANGLE_STRIP / GL_TRIANGLE_FAN
- Pros
  - Very simple
  - Used in many older indie games
- Cons
  - Compressed as bytes to save space, vertices swim and ripple during animation
  - The large collection of models was taken down

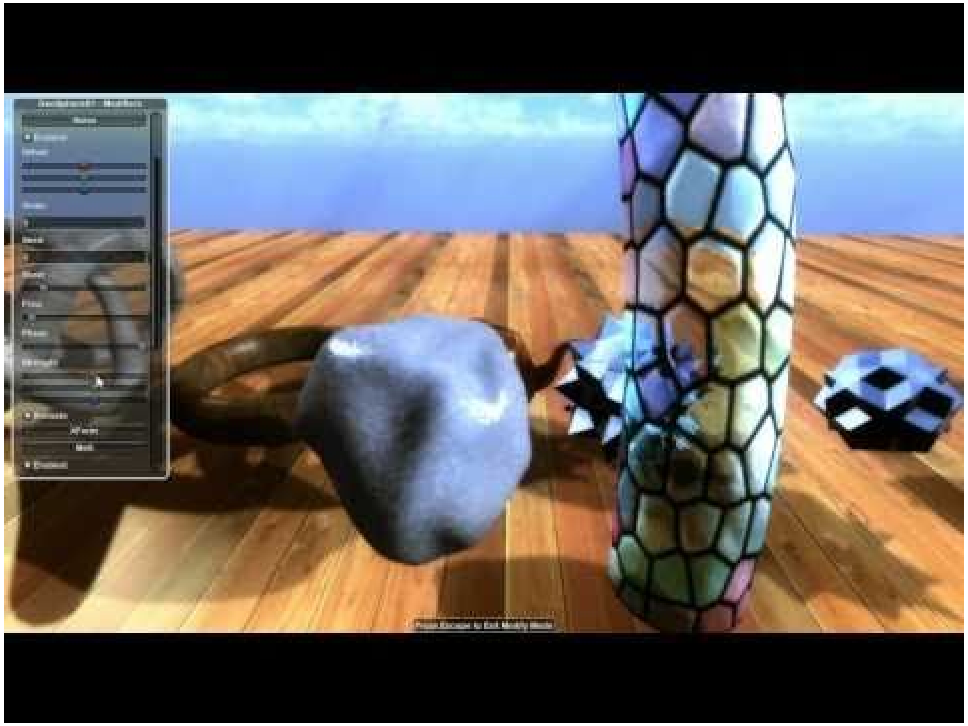
Aside: Modifiers

- Compute vertices from arbitrary function
  - Example: Bend modifier
  - Store the modifier, compute vertices on the fly
- More useful for offline applications
  - Quality vs speed tradeoff



Case Study: Modifiers in Unity3D

- Bend, melt, morph, twist, skew, ripple, ...



http://www.youtube.com/watch?v=GBmgtjyAFaw

3. Skeletal Animation

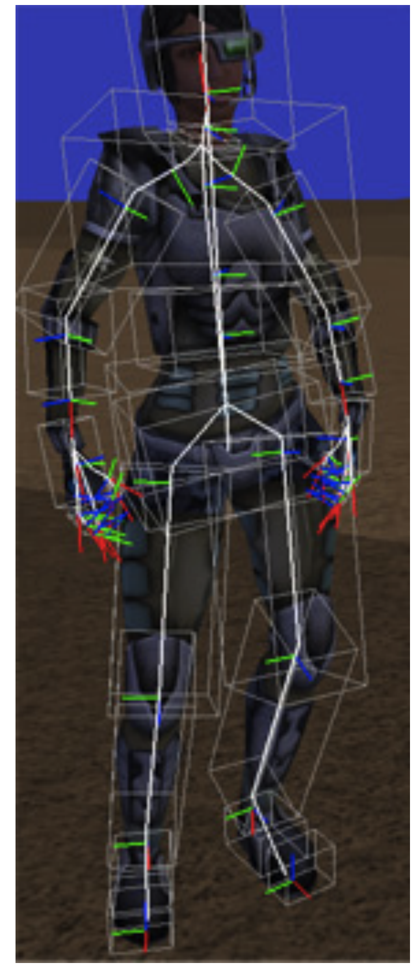
- Most common animation technique for 3D games today
- *Skeleton* composed of *joints* (or *bones*)
  - Each joint has a transformation matrix
  - Skeleton forms a tree of joints
  - Creation of a skeleton: *rigging*
- Pros
  - Much more compact than vertex animation
  - Easier to create animations, just move joints
- Cons
  - Less control over look than vertex animation

Rigid Body Hierarchy Animation

- Hierarchy of 3D primitives
  - Think CS123 scene graph
- Used in Minecraft
  - Implicit scene graph with glPushMatrix() / glPopMatrix()

Skinning

- Process of defining mesh around skeleton
  - Vertices are associated with one or more joints
  - Each association has some weight (from 0 to 1)
- Rendering a vertex
  - Blend between the transformations of its joints
  - Easiest: Linear interpolation
  - State of the art: Dual quaternion blending

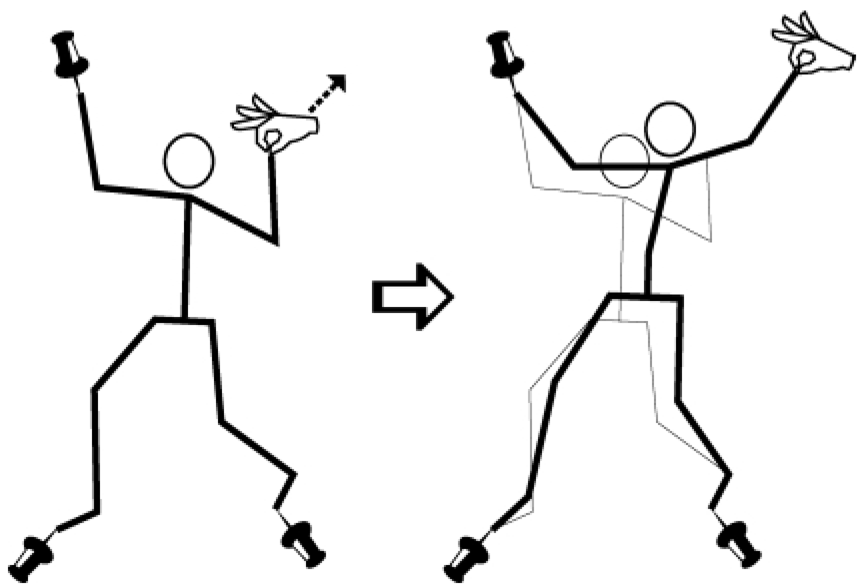


Forward Kinematics

- Compute world-space location given orientation of joints
  - Global matrix of joint depends on parent matrix and local rotation
  - global matrix = parent matrix * local matrix
  - Compute global matrices using DFS over skeleton
- Different ways of representing local matrix
  - Euler angles: 3 angles, rotation about x, y, and z axes
  - Axis-angle: rotation around a vector (quaternions)

Inverse Kinematics

- Compute orientation of joints given world-space location
  - More intuitive user interface
  - Iterative and analytic solutions



Aside: Quaternions

- Quaternions are ~~magic~~ mathematical objects
- A quaternion represents a rotation in 3D
  - Equivalent to a 3x3 matrix with no shear or scale
  - Think of it as an axis and a rotation about that axis
- Nice to work with in several ways:
  - Compact: Stored as a four-vector
  - Easy to interpolate between two rotations
  - Avoids gimbal lock

Rotation by θ about $N = (x, y, z)$ gives a quaternion of $(\cos(\theta/2), x \sin(\theta/2), y \sin(\theta/2), z \sin(\theta/2))$

Rendering a Skin

- Linear blending
  - Transform a copy of the vertex for each joint
  - Linearly interpolate between them using weights
  - Pinching problem for rotations


Rendering a Skin

- Dual quaternion blending
  - State of the art
  - Solves pinching problem
  - Simple GPU-friendly evaluation

Case Study: MD5

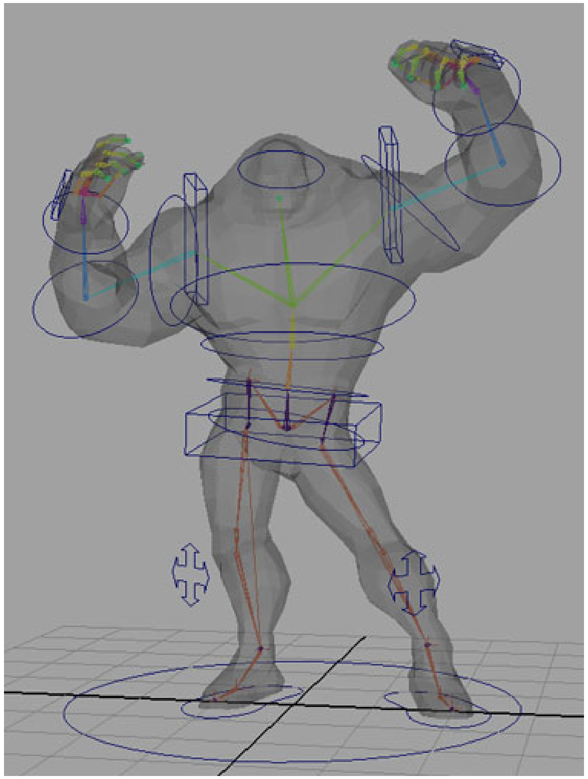
- Used in Doom 3, by id Software (2004)
  - Uses two ASCII md5mesh and md5anim
- md5mesh
  - Mesh and "bind pose" skeleton stored in one file
  - Uses quaternions for joints
  - Stores normalized quaternions using 3 floats
- md5anim
  - Contains an individual animation
  - AABB stored for each frame
- Separation of animation and mesh allows sharing of animations between meshes

Generating Animations

- Set root translation and orientation of joints
- Games use combination of methods
    - Manual animation
    - Procedural animation (inverse kinematics)
    - Ragdoll physics
    - Evolutionary algorithms

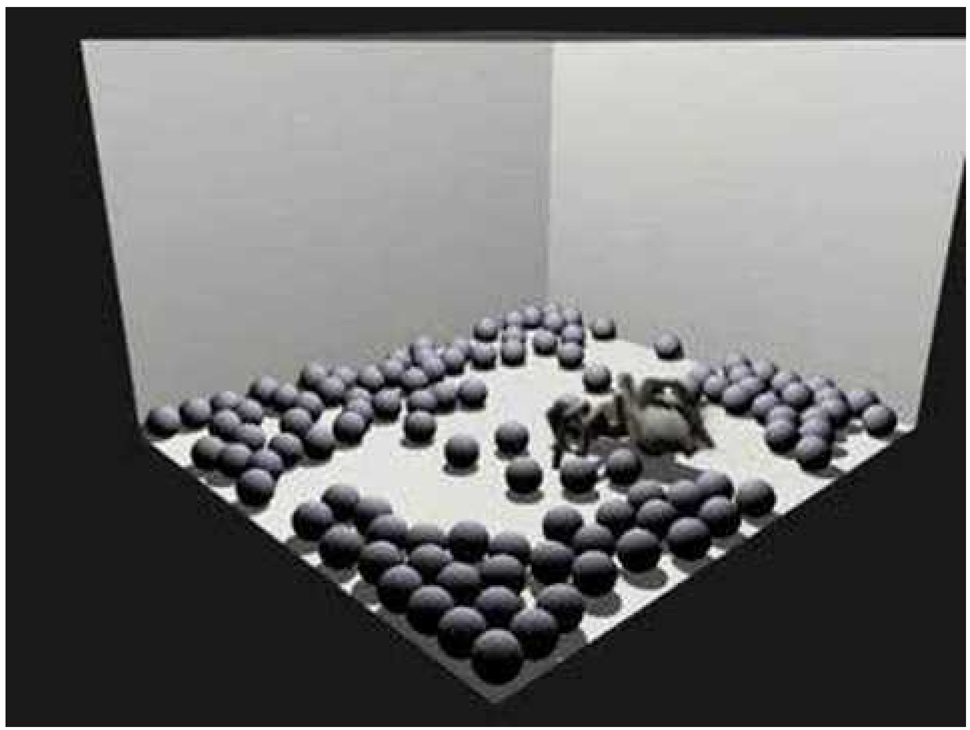
Manual Animation

- Animators specify poses for keyframes on a timeline
  - Splines control interpolation
- Software tools for high-end 3D animation
  - Free: Blender
  - Nowhere near free: Maya, 3ds Max
- Neither of us are actually good at this
  - To learn from people who are, take CS125/CS128!



Procedural Animation

- Real-time procedural animation of spiders



http://www.youtube.com/watch?v=I1P_B65XW4I

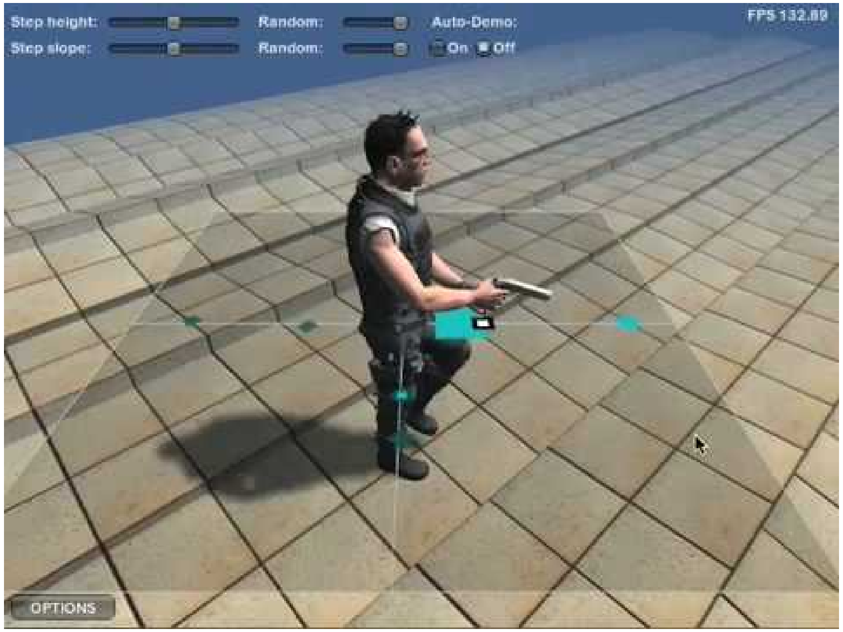
Procedural Animation

- Vehicles in Unreal Tournament 3 (2007)

http://www.youtube.com/watch?v=xLr419lmRBE

Case Study: Locomotion

- Unity library for realistic movement
  - Sets pose of skeleton procedurally
  - Blends between animations: up vs down slopes
  - Places feet tangent to surface



http://www.youtube.com/watch?v=v2q5kuic6HA

Ragdoll Physics

- Blending of physics and animation
- Rigid bodies tied together by constraints
  - Typically spring-damper constraints
- Many variations on standard approach:
  - Verlet integration: Model each bone as a point
  - Blended ragdoll: Use preset animation, but constrain output based on a model of a physical system
    - Used in Halo 2, Halo 3, Left 4 Dead, etc.

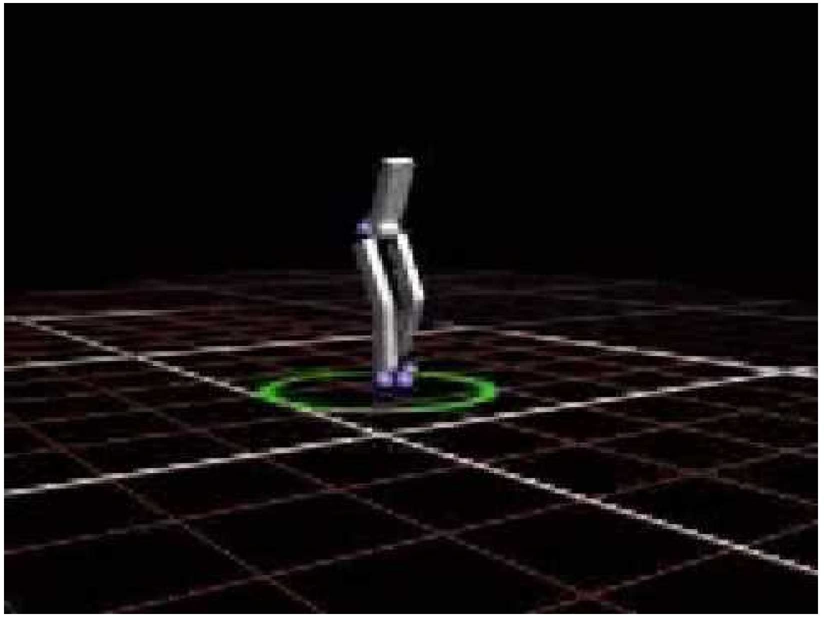
Case Study: Euphoria & Endorphin

- Ragdoll physics engine & animation system
- Used in many recent games
  - *Red Dead Redemption*, *Star Wars: Force Unleashed*

http://www.youtube.com/watch?v=Ae3fgj2x1aI

Evolutionary Algorithms

- Evolve animation to meet a goal
- Goal defined by fitness function:
  - Move at a certain speed
  - Minimize energy used



http://www.youtube.com/watch?v=JFJkpVWTQVM

Mesh Formats

- Range of file format capabilities
- Wavefront OBJ
  - Extension: *.obj
  - Simple, easily parsable ASCII format
  - Doesn't support animation or multitexturing
- COLLADA: COLLAborative Design Activity
  - Extension: *.dae
  - Used for transferring models between applications
  - XML format general enough to define any model
  - Includes lights, materials, cameras, physics models

Mesh Formats

- All major game engines use their own format
  - Games are high-performance applications with custom requirements
  - 3D models integrate tightly with engine-specific resource loading systems (i.e. data archives)
- Finding free game assets is difficult
  - Especially for animated models
  - Open Asset Import Library: Open source C++ library that imports and exports many different formats

Sources of Free 3D Models

- http://opengameart.org/
- http://turbosquid.com/
- http://sketchup.google.com/3dwarehouse/

Minecraft Week 3

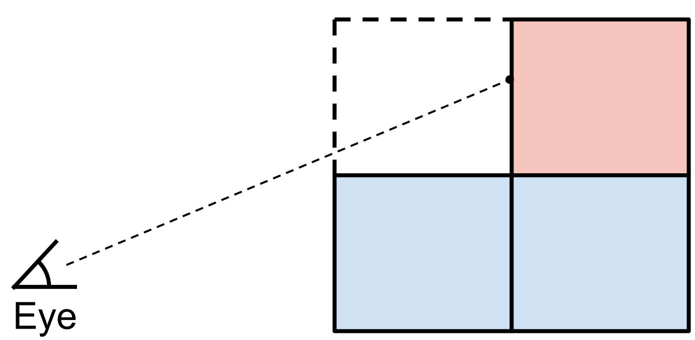
Minecraft: Week 3

- Adding and removing blocks
  - Use voxel traversal for rays
- Enemies with simple movement and jumping
  - Collide as AABB, must collide with the player

- Streaming chunks
  - Don't need to save modifications to the world

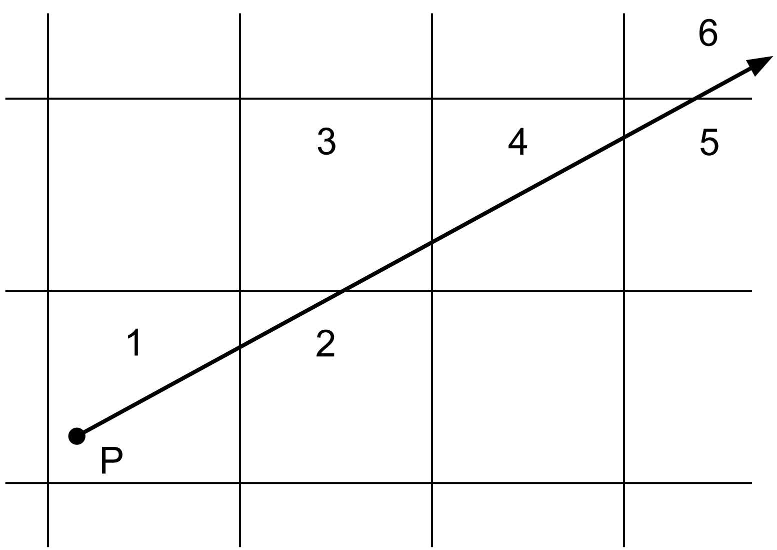
Adding and Removing Blocks

- Cast a ray from the player's eye
- First collision point determines block to remove
- Block adjacent to face of collision determines block to add
- Make sure not to add to a block occupied by the player!
- How to efficiently intersect the ray with a large number of blocks? Voxel Traversal!



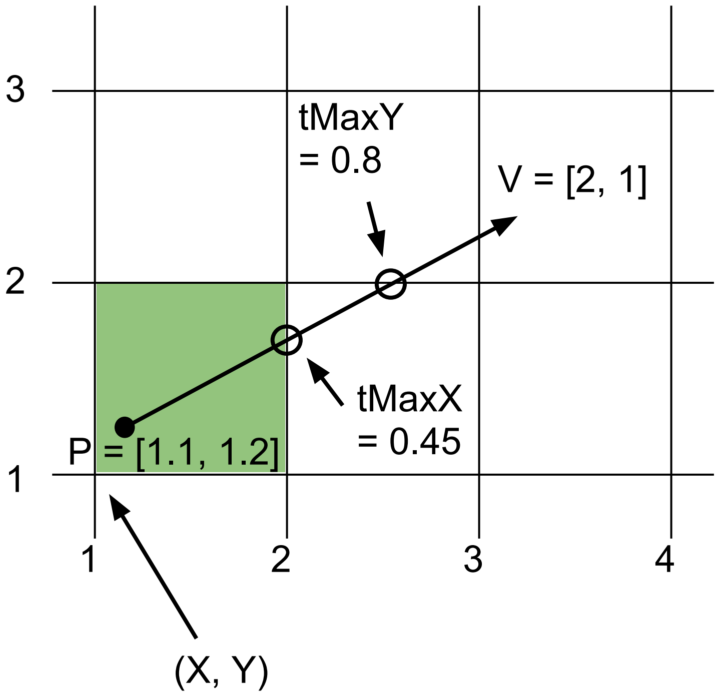
Fast Voxel Traversal Algorithm

- Ray equation: *P(t) = P(0) + tV*
- Two stage algorithm: initialization and traversal
- Intuition: break ray into components based on grid boundaries



Fast Voxel Traversal Algorithm

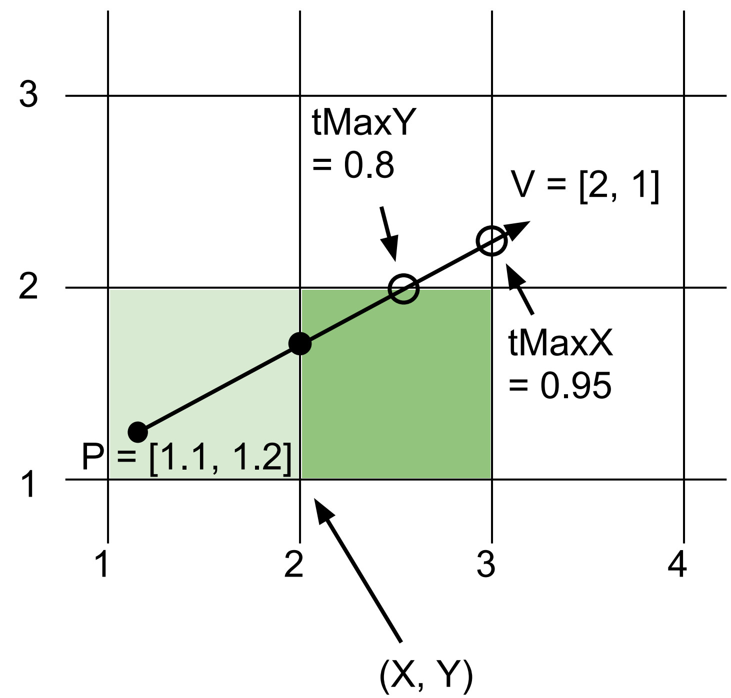
```
int X, Y;                  // integer block coordinates
int stepX, stepY;          // +1 or -1 depending on direction
float tMaxX, tMaxY;        // t value to next integer boundary
float tDeltaX, tDeltaY;    // t delta to span an entire cell

loop {
  ProcessVoxel(X, Y);
  if(tMaxX < tMaxY) {
    tMaxX += tDeltaX;
    X += stepX;
  } else {
    tMaxY += tDeltaY;
    Y += stepY;
  }
}
```



Fast Voxel Traversal Algorithm

```
int X, Y;                  // integer block coordinates
int stepX, stepY;          // +1 or -1 depending on direction
float tMaxX, tMaxY;        // t value to next integer boundary
float tDeltaX, tDeltaY;    // t delta to span an entire cell
```

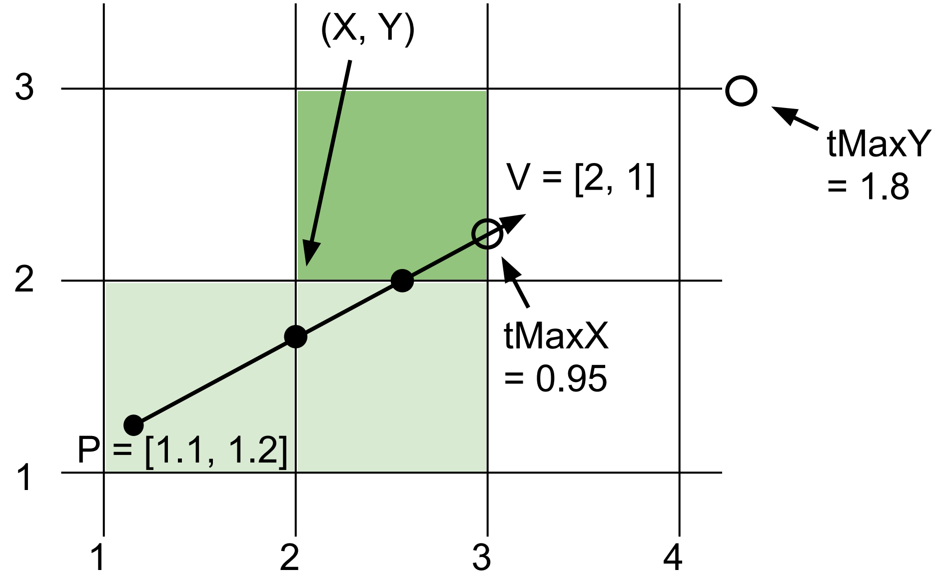
```
loop {
  ProcessVoxel(X, Y);
  if(tMaxX < tMaxY) {
    tMaxX += tDeltaX;
    X += stepX;
  } else {
    tMaxY += tDeltaY;
    Y += stepY;
  }
}
```



Fast Voxel Traversal Algorithm

```
int X, Y;                   // integer block coordinates
int stepX, stepY;           // +1 or -1 depending on direction
float tMaxX, tMaxY;         // t value to next integer boundary
float tDeltaX, tDeltaY;     // t delta to span an entire cell
```

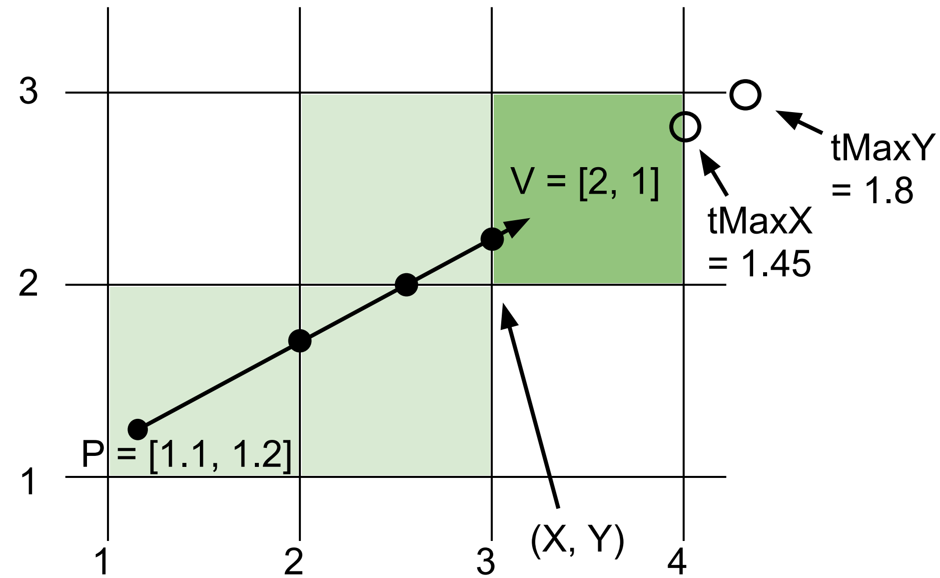
```
loop {
  ProcessVoxel(X, Y);
  if(tMaxX < tMaxY) {
    tMaxX += tDeltaX;
    X += stepX;
  } else {
    tMaxY += tDeltaY;
    Y += stepY;
  }
}
```



Fast Voxel Traversal Algorithm

```
int X, Y;                  // integer block coordinates
int stepX, stepY;          // +1 or -1 depending on direction
float tMaxX, tMaxY;        // t value to next integer boundary
float tDeltaX, tDeltaY;    // t delta to span an entire cell
```

```
loop {
  ProcessVoxel(X, Y);
  if(tMaxX < tMaxY) {
    tMaxX += tDeltaX;
    X += stepX;
  } else {
    tMaxY += tDeltaY;
    Y += stepY;
  }
}
```



Fast Voxel Traversal Algorithm

- Can be easily extended to 3D
  - 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

Enemies

- Keep behavior simple
  - e.g. choose a random direction biased towards the player and jump if on the ground
- Reuse AABB-world collision detection from last week
- Collide with player using AABB-AABB collision detection
  - Don't need to move out of collision
- Optional: Make a rigid-body hierarchy with animation

Aside: Random Sampling

- We want a random vector biased towards the player
  - Method 1: Linearly combine the normalized vector facing the player with a uniformly random unit vector
  - Method 2: Sample from a non-uniform random distribution

Aside: Random Sampling

- How do we get a uniformly random 3D unit vector?
- What doesn't work
  - Choose two random angles using spherical coordinates, then convert to Cartesian coordinates
    - More points at poles of the sphere
  - Choose uniformly random x, y, and z values and normalize
    - Sampling over a cube instead of a sphere
- What works
  - Choose an angle θ from [0, 2 pi] and z from [-1, 1] (point on a cylinder), then map onto a unit sphere:
  - [x, y, z] = [sqrt(1 - z^2)cos(θ), sqrt(1 - z^2)sin(θ), z]

Aside: Random Sampling

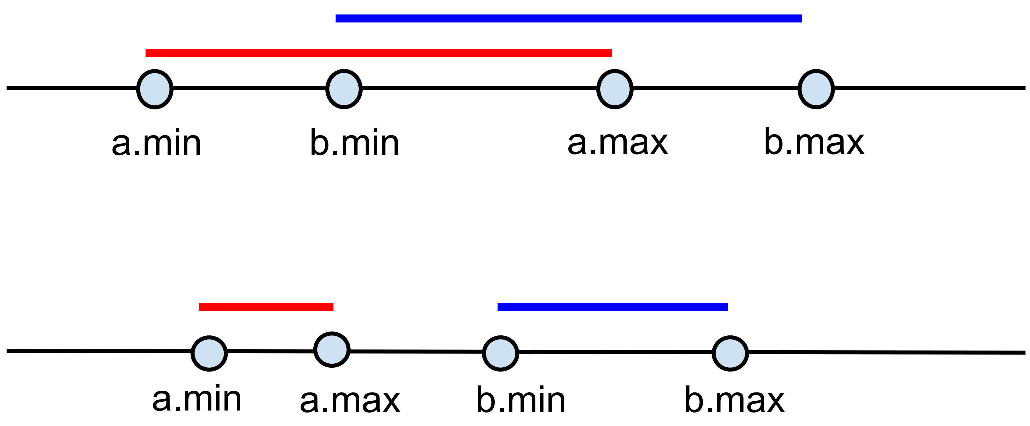
- C++ only provides `rand()` for pseudo-random ints
  - Convert to [0, 1] float/double by dividing by `RAND_MAX`
  - Random vector implemented in support code already: `Vector3::randomDirection()`
- Upcoming C++11 standard library has three different pseudo-random number generators
  - Also has non-uniform random distributions
    - e.g. poisson distribution, gamma distribution

AABB-AABB Collision Detection

- Simple interval test along each axis
- Return true only if each interval test is true
- Most efficient with min-max representation of AABB

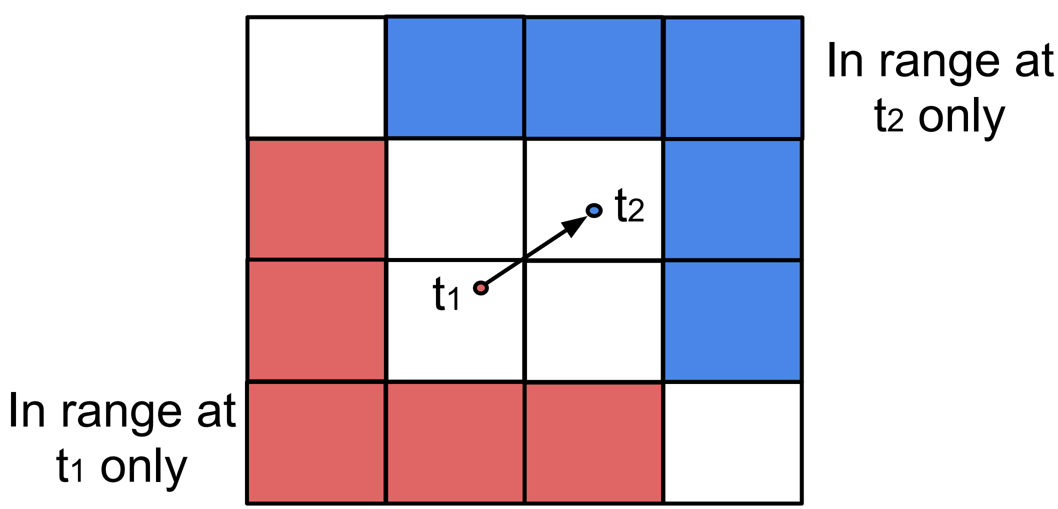
Interval test:

```
return a.max >= b.min && b.max >= a.min;
```



Streaming

- Only store chunks within a certain distance from the player
  - AABB or sphere around player
- Update when player moves between chunks
  - Remove blocks that went out of range, add blocks that came into range



Streaming

- 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 >50 chunks
  - Too much work for a single game update
- Simplest solution: queue of added chunks
  - Dequeue one chunk per world update
  - Build chunk's vertex buffer when it's taken off the queue
- More complicated solution: multi-threaded system

Saving and Loading

- With chunk streaming, modifications to a chunk 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 them stream in
- How to efficiently save and load so much data?

Saving and Loading: Minecraft

- Only save modified chunks
  - Otherwise too much to store
- Filesystem for storing world information
  - Each 32x32 group of chunks put in a "region" file
  - Region files named "r.[x].[z].mcr", where x and z are the index of the region
  - 8KiB header with info on which chunks are present, when they were updated, and where they can be found
- Chunk data compressed with ZLib
- More information: http://www.minecraftwiki.net/wiki/Beta_Level_Format

C++ Tip of the Week

- Virtual destructors
  - Needed for classes that have subclasses with destructors

```
struct GameObject {
  ~GameObject() {}
};

struct Player : GameObject {
  int *buffer;
  Player() : buffer(new int[1024]) {}
  ~Player() { delete [] buffer; }
};

GameObject *player = new Player();
delete player; // buffer is leaked because ~Player() isn't called!
```

C++ Tip of the Week

- Virtual destructors
  - Needed for classes that have subclasses with destructors

```
struct GameObject {
  virtual ~GameObject() {}
};

struct Player : GameObject {
  int *buffer;
  Player() : buffer(new int[1024]) {}
  ~Player() { delete [] buffer; }
};

GameObject *player = new Player();
delete player; // ~Player() is called and buffer is deleted
```

Weeklies

References

Amantides, J., Woo, A. "A fast voxel traversal algorithm for ray tracing". *Eurographics*, G. Marechal, Ed. Elsevier North-Holland, New York, 1987, 3-10.

Ericson, Christer (2005). *Real Time Collision Detection*. Boston, MA: Morgan Kaufmann Publishers.