Class 5

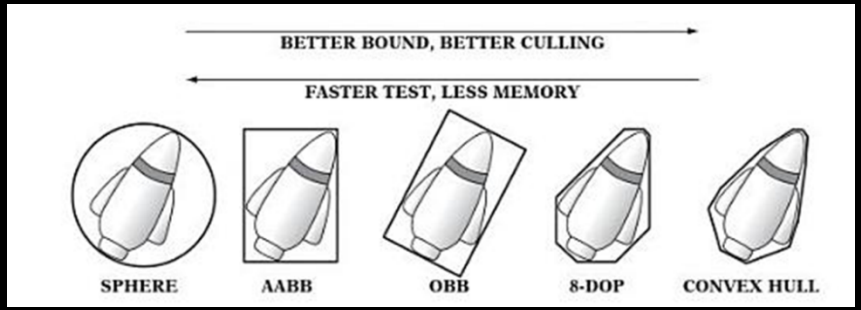
Engine Optimizations

Spatial organization

- Standard collision detection is $O(n^2)$
  - Too slow for large game worlds
- Solution: spatially organize to discard far-away data
  - Other game objects
  - Pieces of static geometry
- Many approaches
  - Bounding volumes
  - Bounding volume hierarchy
  - Uniform grid
  - Hierarchical grid
  - Octree
  - K-D Tree
  - BSP tree

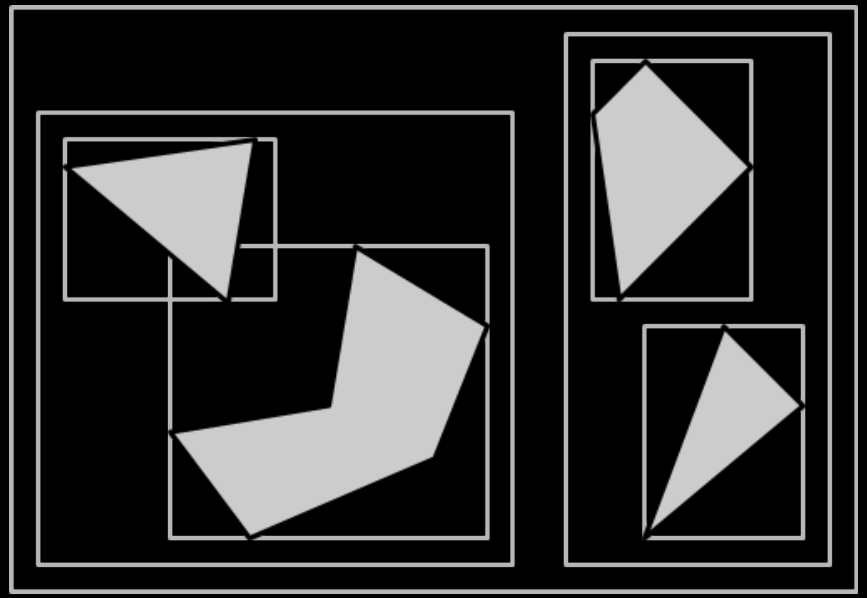
Bounding volumes

- Wrap objects in simple geometry (bounding volumes) to speed up collisions
  - If objects are far apart, their bounding volumes won't collide, and the detailed test is not needed
- Doesn't solve $O(n^2)$ runtime alone



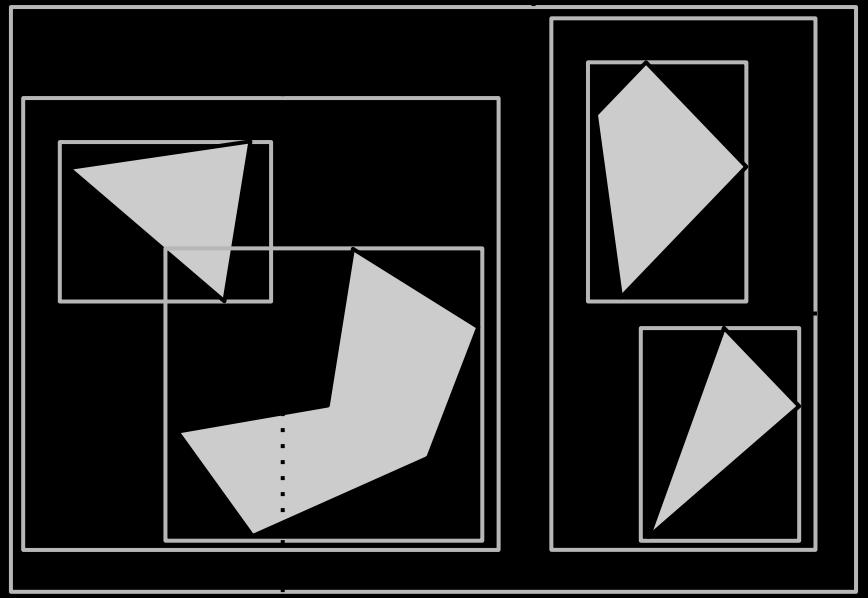
Bounding volume hierarchy (BVH)

- Parent volumes contain child volumes
- Sibling volumes may overlap (not spatial partitioning)
- Speeds up collisions between dynamic entities
- Also speeds up rendering



BVH construction

- Top-down construction easier than bottom-up
- Start with a volume containing all objects
- Find a partition via some heuristic
- Sort objects into two sub-volumes and discard partition
- Repeat recursively



BVH construction

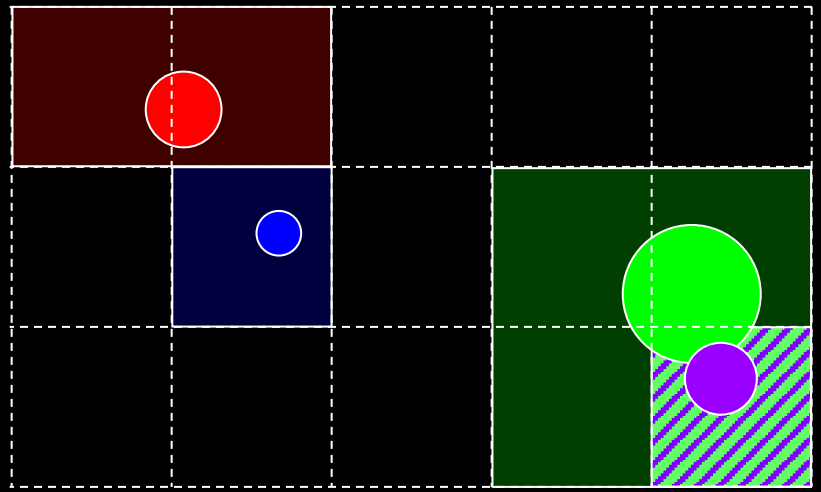
```
buildBVH(node, objects):
        // fit bounding volume of node to objects
        ...
        if numObjects <= minObjectsPerLeaf:
                node.objects = objects
        else:
                // partition objects according to a heuristic
                ...
                buildBVH(node.left, leftObjects)
                buildBVH(node.right, rightObjects)
```

BVH construction heuristics

- Difficult to choose good partitioning axes
- Goals
  - Minimize node volume
  - Keep branches balanced
  - Minimize sibling overlap
- Strategies
  - Median-cut (divide into equal-size parts with respect to a selected axis)
  - Minimize sum of child volumes
  - Maximize separation of child volumes

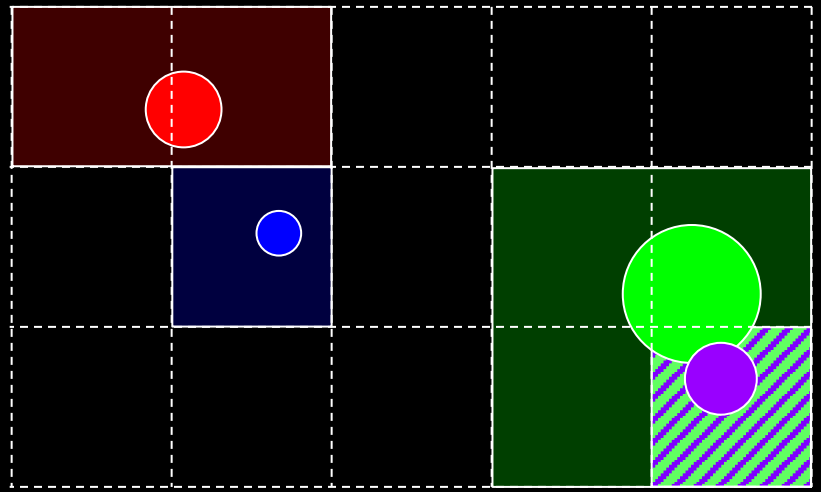
Uniform grid

- Speeds up collisions between dynamic entities
- Axis-aligned cube cells (voxels)
  - Each cell has list of objects
- May be dense (array) or sparse (hash-based)
- Each frame: update cells for objects that moved



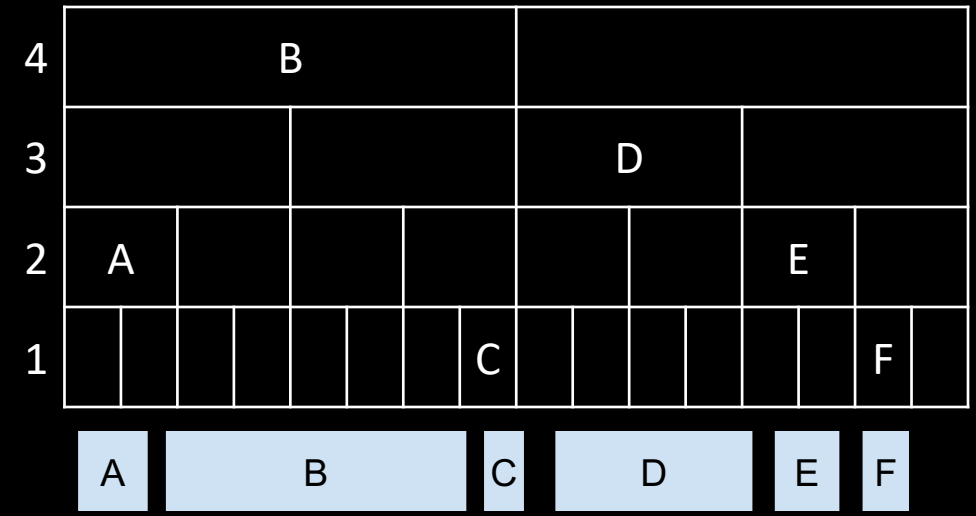
Uniform grid

- Pros
  - Simple to generate and modify
  - Good for static and dynamic objects
  - O(1) access to neighbors
- Cons
  - Not appropriate for objects of greatly varying sizes
  - Dense representation wastes memory in empty regions



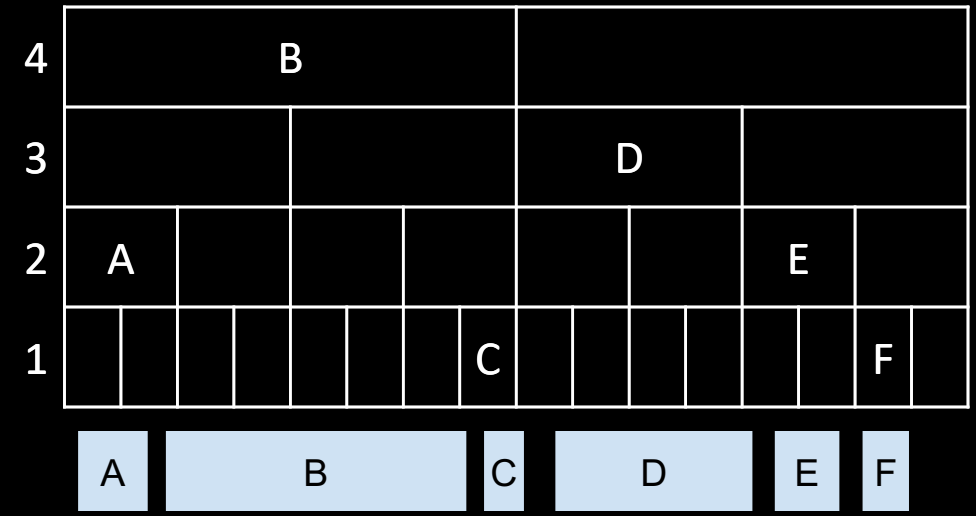
Hierarchical grid

- Hierarchy of differently-sized uniform grids
  - N levels, cells at top level are big enough to cover bounding box of largest game object
  - Each sub-level's cells are half the scale
- Inserting objects
  - Find the level where cells are big enough to fit it fully
  - In that level, insert into the cell that contains its center
- 1D example on right



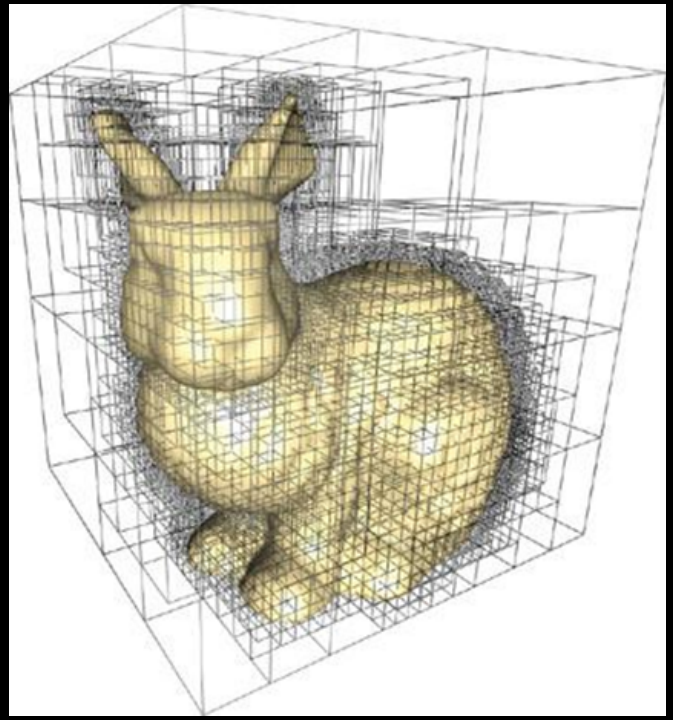
Hierarchical grid queries

- Traverse over all levels
- Test cells containing object's bounding box and neighboring cells
- 1D example: colliding object E



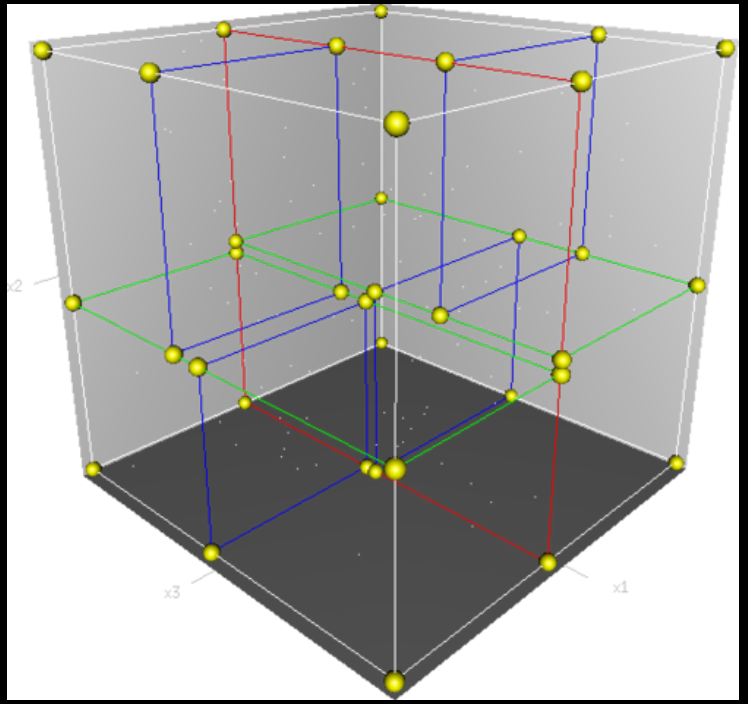
Octree

- Axis-aligned hierarchical partitions
- Hierarchical grid with one top-level cell
- Each node is split into eight child nodes
- Pros:
  - Simple construction for static scenes
  - Easy collision tests: all AABBs
- Cons
  - Traversing tree is expensive if the tree is deep or unbalanced

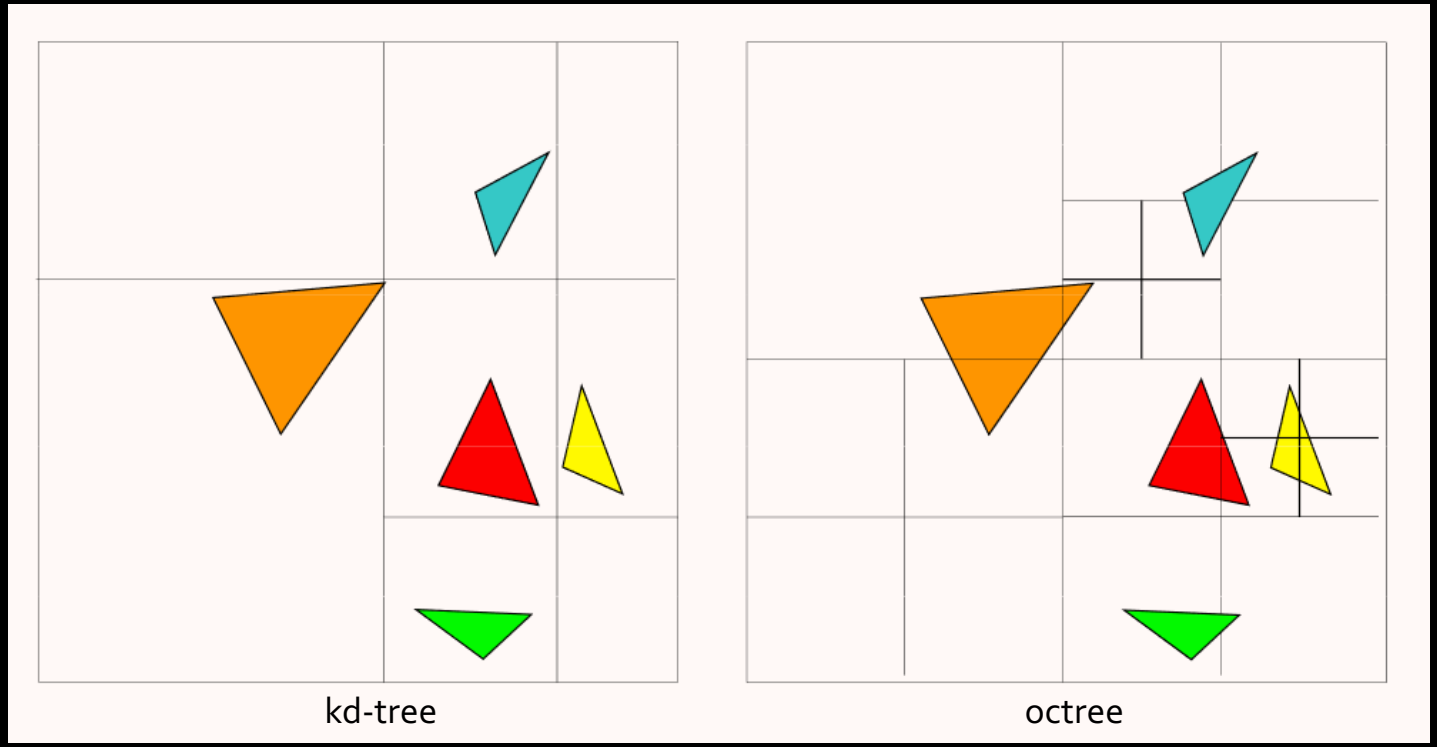


K-D Tree

- Axis-aligned binary tree
- At each node we select one axis to split along (usually we cycle through the dimensions down the tree)
- Need to use some heuristic like median-cut in order to choose where we split
- Special case of BSP trees (up next)

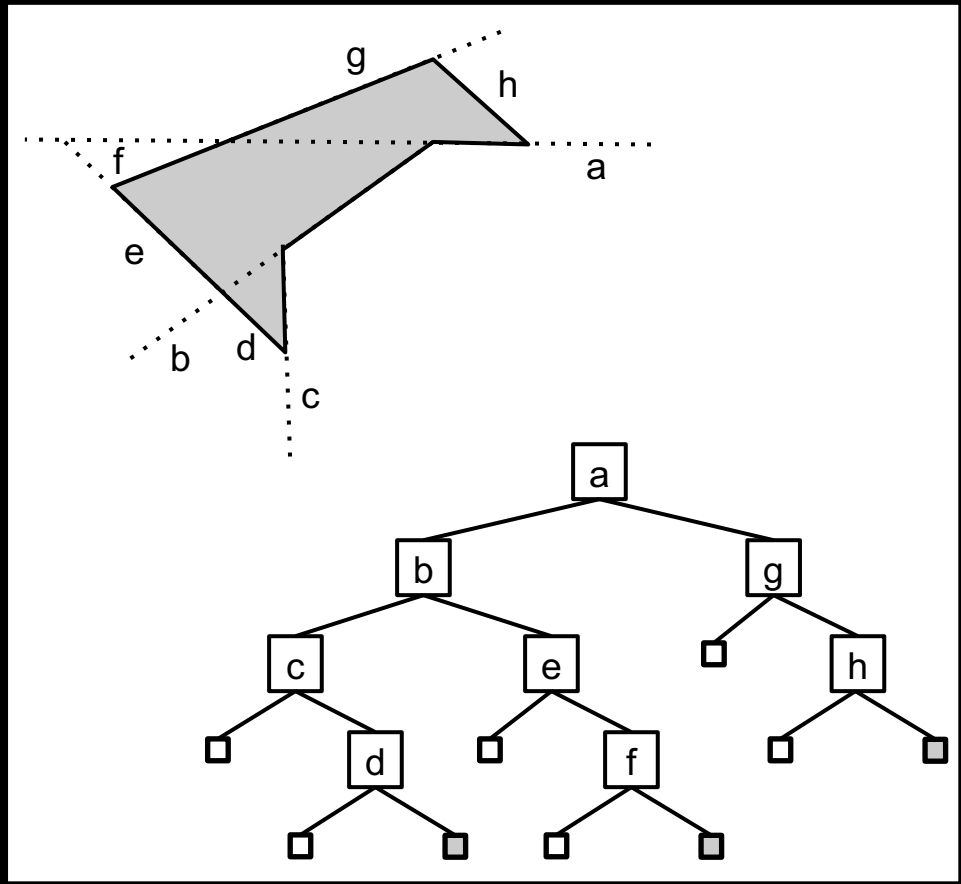


K-D Tree vs. Octree



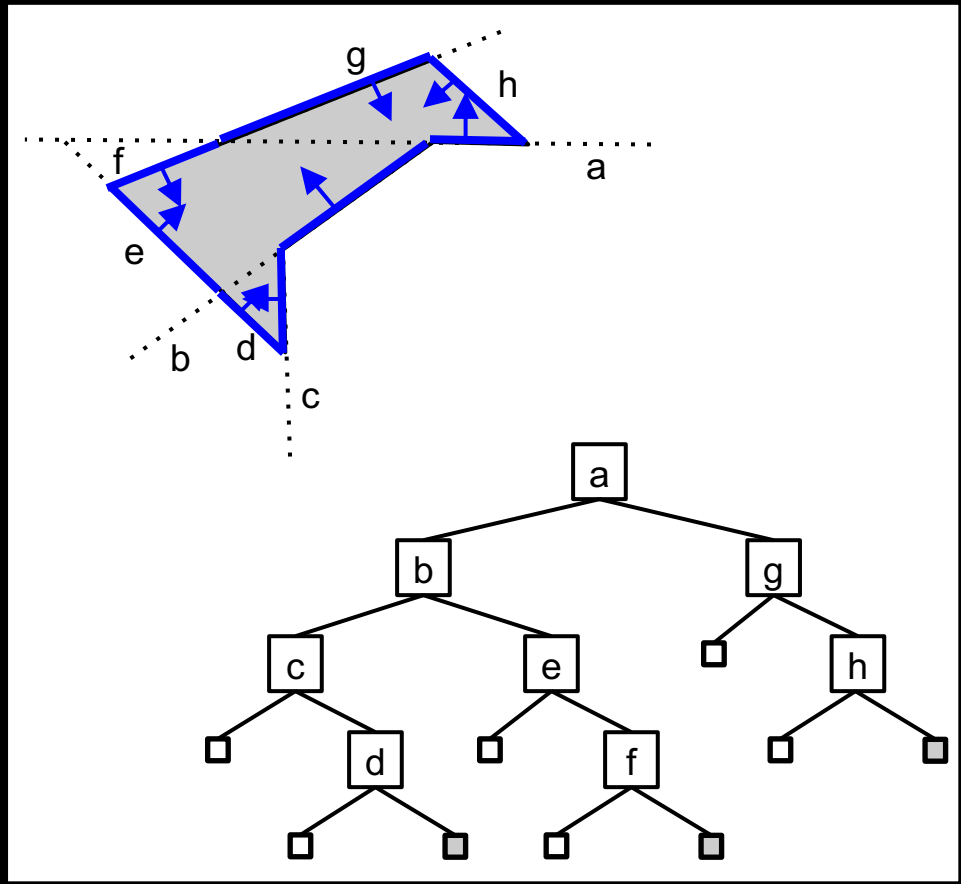
BSP Tree

- Binary Space Partitioning Tree
- Hierarchy of planar half-spaces (not necessarily axis-aligned like in a k-d tree)
  - Each node has a plane, divides the space of the parent in two
  - Models solid vs. empty space
  - Leaves represent convex polytope
    - Some solid, some empty
    - Some have infinite area/volume
- Works best for indoor environments (flat, man-made surfaces)
- Used by original Doom and Quake
  - Still used today (Source, Unreal, Call of Duty IW engine)



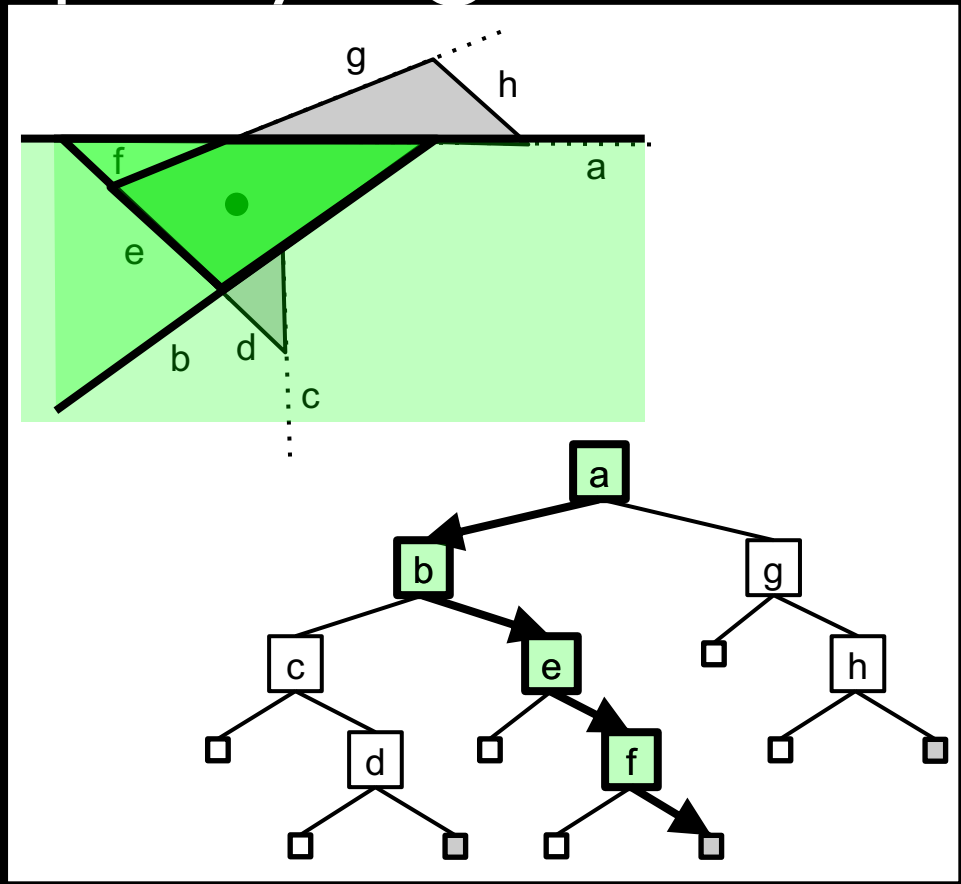
BSP tree construction

- Recursively pick a triangle according to a heuristic, split scene along that plane
- Tree needs to be balanced for good performance



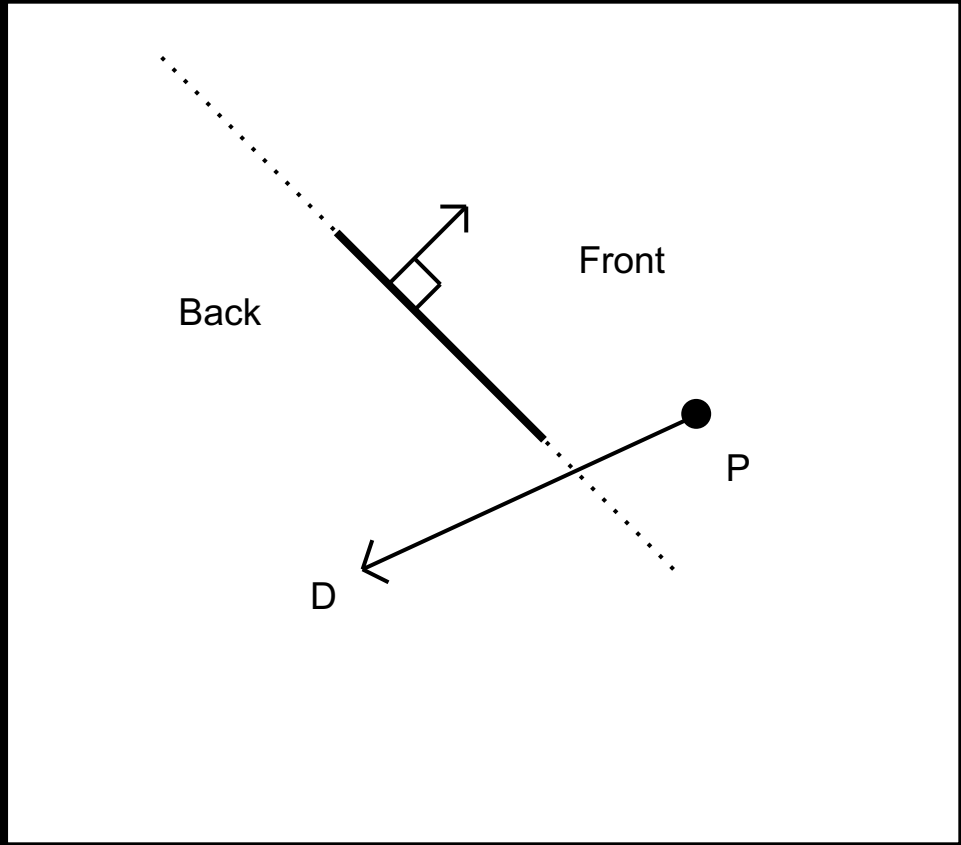
BSP tree querying

- To test if a point is inside or outside:
  - Start at root
  - Visit front child if point in front of plane, otherwise visit back child
  - If the point is in front of a leaf node, it is outside, otherwise it is inside



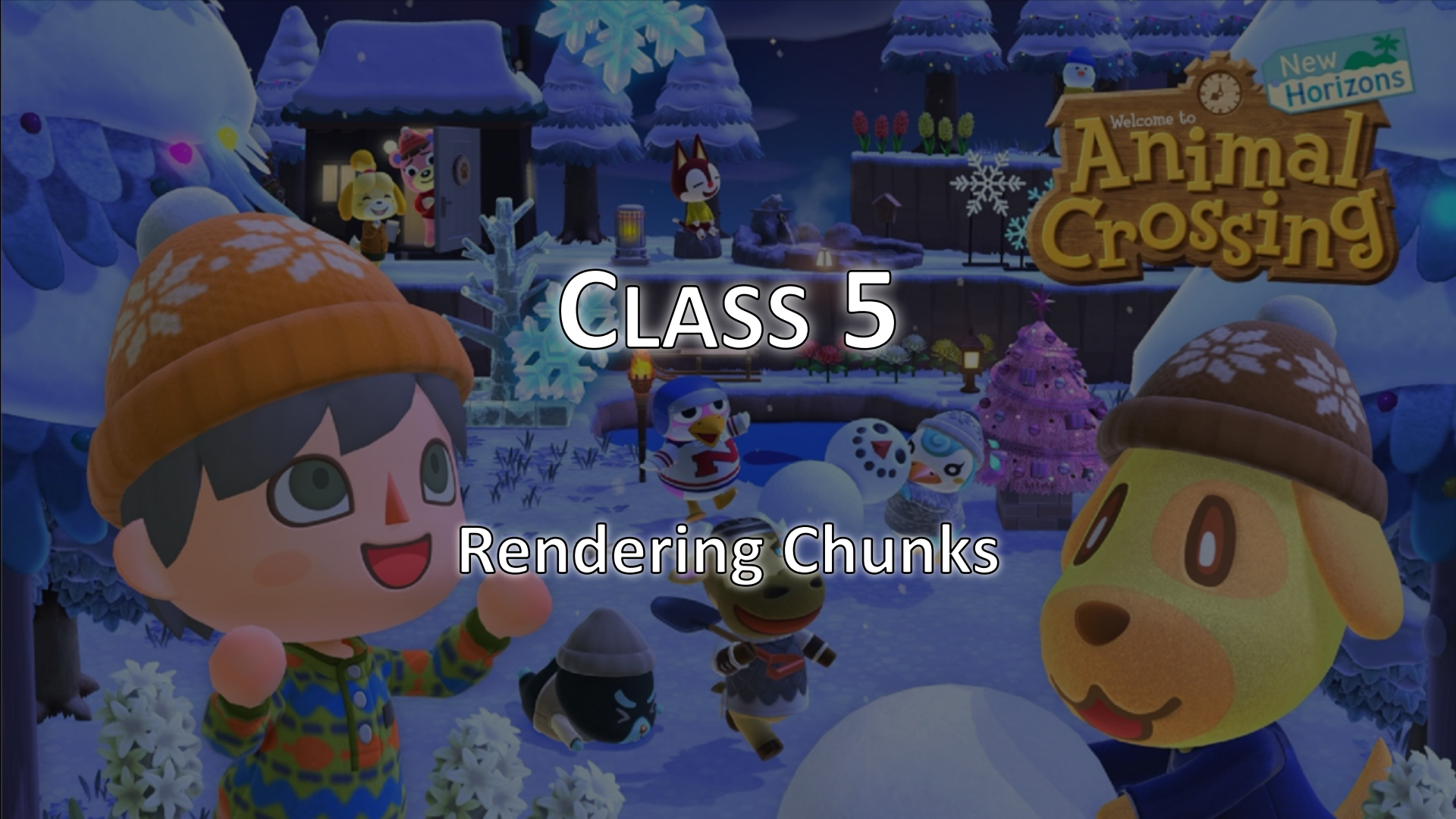
BSP tree ray traversal

- Given ray (P,D) and root
- If P is behind plane
  - Recursively traverse back node
  - Hit-test polygons on plane
  - Recursively traverse front node if $D \cdot N > 0$
- If P is in front of plane
  - Recursively traverse front node
  - Hit-test polygons on plane
  - Recursively traverse back node if $D \cdot N < 0$



BSP tree pros & cons

- Pros:
  - Because it's a binary tree, most tests O(log n)
  - Easy collision detection and raycasting
- Cons:
  - Very expensive to build (optimal is NP-complete!)
  - Not suitable for dynamic objects
  - Numeric stability can be tricky

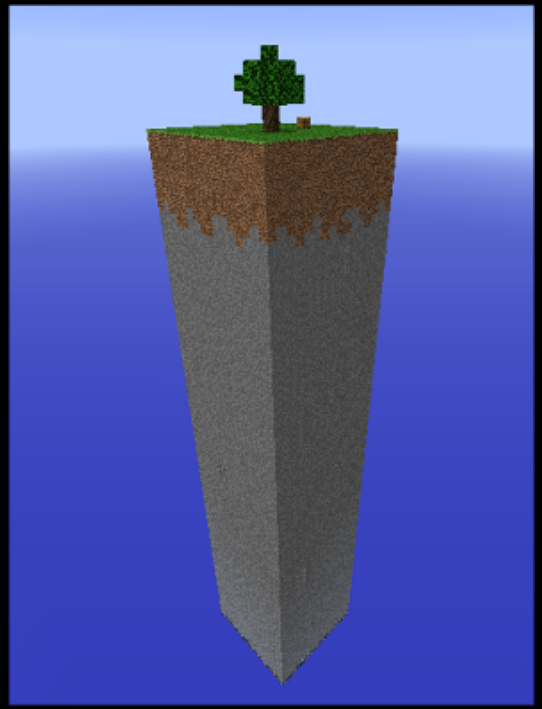


Class 5

Rendering Chunks

Drawing Things

- We have a ton of walls and floors and roofs, so how should we go about drawing them?



Attempt #1

- We'll loop through each wall, set the material, and then draw each wall individually.
- But each time we draw a wall, we have to send information from the CPU over to the GPU.

Attempt #1

- For each of these walls, a draw call needs to travel to the GPU for that wall to be rendered
  - And this would happen on every frame
  - Sending data to the GPU is a huge bottleneck for drawing

Attempt #2: Storing Shapes

- Solution: Create and store shape for entire pieces of static geometry
  - In our case, we'll create one Shape **per chunk**
- Pack everything you need into a single Shape and draw it once
- Only a single draw call per chunk

Why bother?

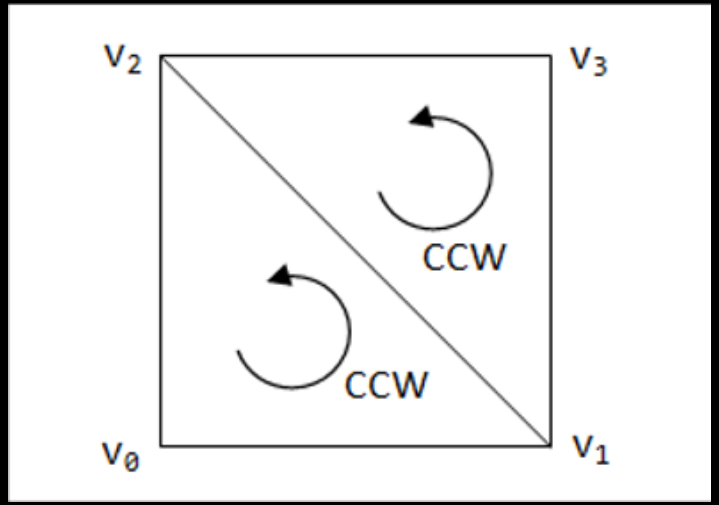
- While fairly time-consuming to set up, the speed increase is incredible
  - Only a single draw call
  - Don't have to change texture for each block (even if they should be colored differently)
    - More on this later

How to do it

- Create each chunk shape:
  1. Create a Shape (see helper classes)
  2. Generate the Shape based on the chunk's walls
  3. Store the Shape
- When drawing, iterate over each chunk:
  1. Draw the Shape

Generating the Shape (Faces)

- For each face, need to specify:
  - Vertices of that face
  - Triangles of that face

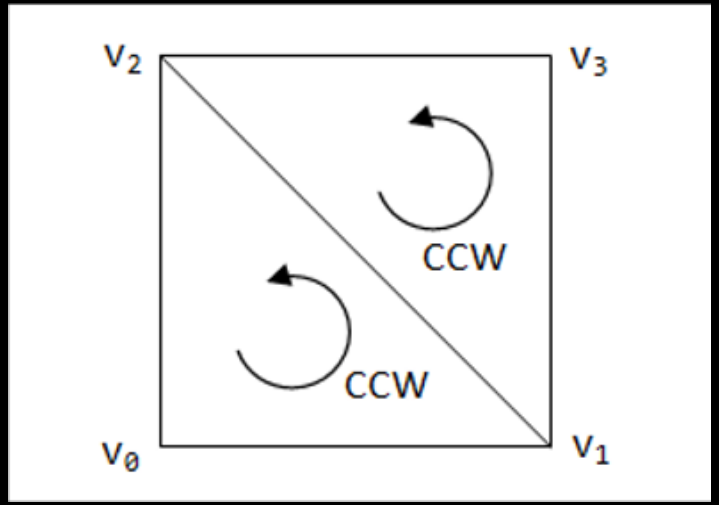


Generating the Shape (Vertices)

- For each vertex, you need to specify:
  - Position (self-explanatory)
  - Normal (the perpendicular to the cube face)
  - Texture coordinates (more on this next)
- Store all vertices in a vector of floats
  - 8 floats total per vertex
    - 3 for position
    - 3 for normal
    - 2 for texture
  - 4 vertices total per face
    - It's a quad

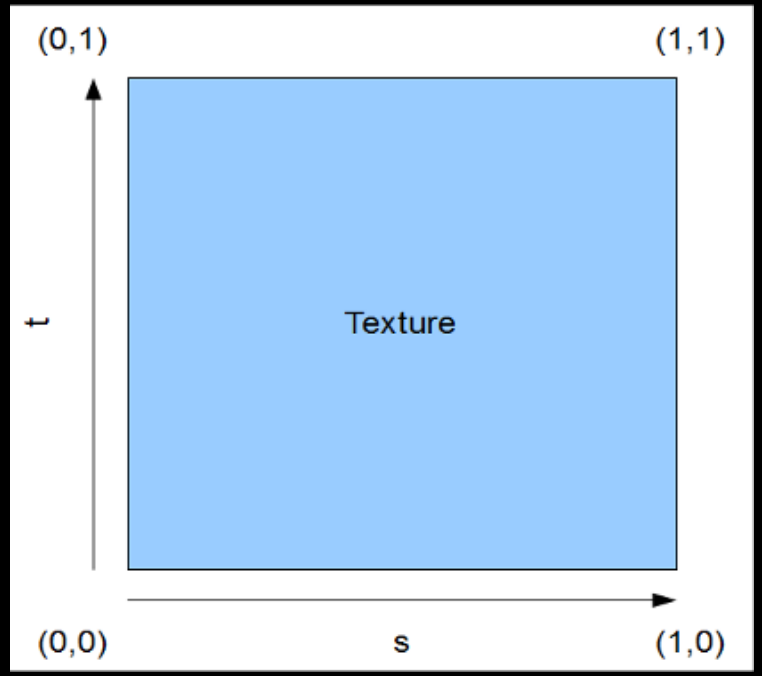
Generating the Shape (Triangles)

- Store all triangles in a vector of ints
  - 3 ints per triangle
    - Specify vertices in counter-clockwise order
    - Int corresponds to position of the vertex in the vertex array
  - Two triangles per face
    - It's a quad



Textures

- Quick recap …
- Textures are basically just images
- Can use “texture coordinates” to specify what part of an image to texture a triangle with
  - (0.0, 0.0) corresponds to upper left of image
  - (1.0, 1.0) corresponds to lower right of image
- We specify the “texture coordinates” for vertices of triangle
    - Texture coordinates for in between points interpolated between these



Texture Atlasing

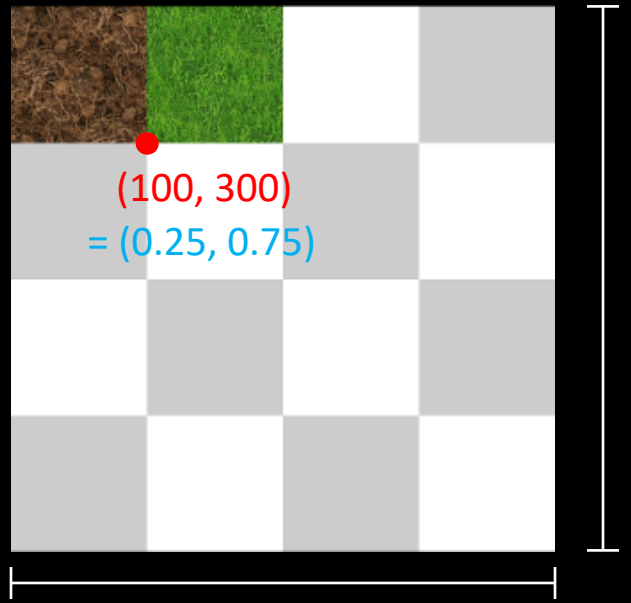
- When rendering chunks, we bind a single image (the texture atlas) which is used to texture all of the terrain
- Can specify texture coordinates for each face individually
- The texture coordinates are defined such that they map to subimages of the atlas
- ~10 fps boost (compared to binding an unbinding individual images)

Texture Atlasing

- You need to know the dimensions of your texture atlas first
  - Maybe the size of the textures too (if they're uniformly sized)
  - Ours is a 256x256 image of 16x16 textures
- Subimages will likely be specified in pixels
- So we need to convert pixel positions to OpenGL texture coordinates

Coordinate Conversion

- To convert pixel coordinates to OpenGL texture coordinates:
  - $(x, y) \rightarrow (\frac{x}{size}, \frac{y}{size})$
- Assume the same origin for both coordinate systems
- Example: convert point at bottom-left of grass
  - Texture size is 400x400
  - Point is at (100, 300)
  - $\left(\frac{100}{400}, \frac{300}{400}\right) \rightarrow (0.25, 0.75)$



Pseudocode

For each chunk:

- Initialize the following:
  - A vector of floats that c*ould* hold ALL of your vertices
  - A vector of ints that can hold all of your triangles
  - A counter to keep track of the number of vertices
  - A Shape to hold the chunk's shape
- For each wall:
  - Is the wall visible? If so, add all vertices and triangles to your array, increment counter
  - Otherwise, skip the wall
- Repeat for floors and ceilings
- Create a shape using the vertices and triangles
  - std::shared_ptr<Shape> shape = std::make_shared<Shape>(vertices, triangles);

Tips

- Remember to use counter-clockwise order for triangle vertices!

Class 5

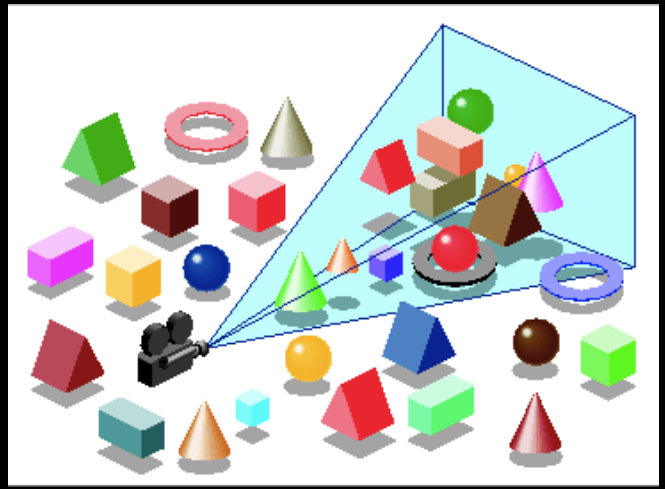
Frustum Culling

Frustum Culling

THE VIEW FRUSTUM

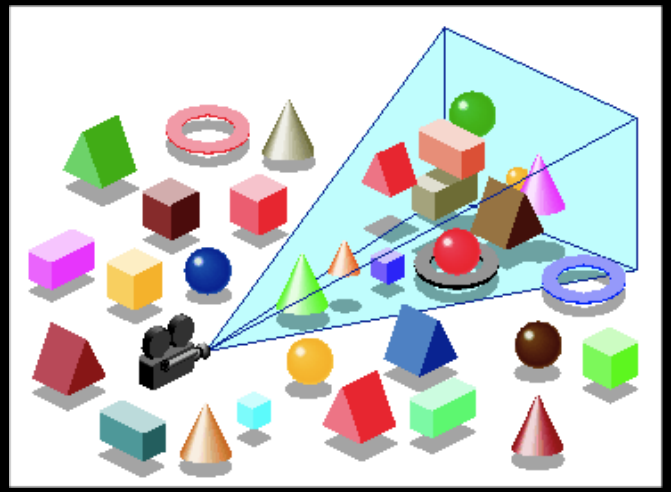
What is it?

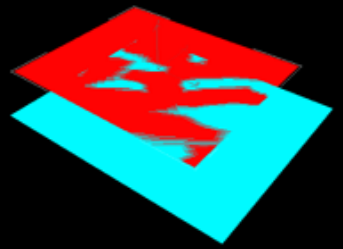
- The volume of world objects that can actually be seen by the camera
- Shaped like a pyramid, bounded by:
  - Far plane (the “base” of the frustum)
  - Near plane (the “cap” of the frustum)
  - Field of view/viewport size (determine the “walls” of the frustum)



What we're doing now…

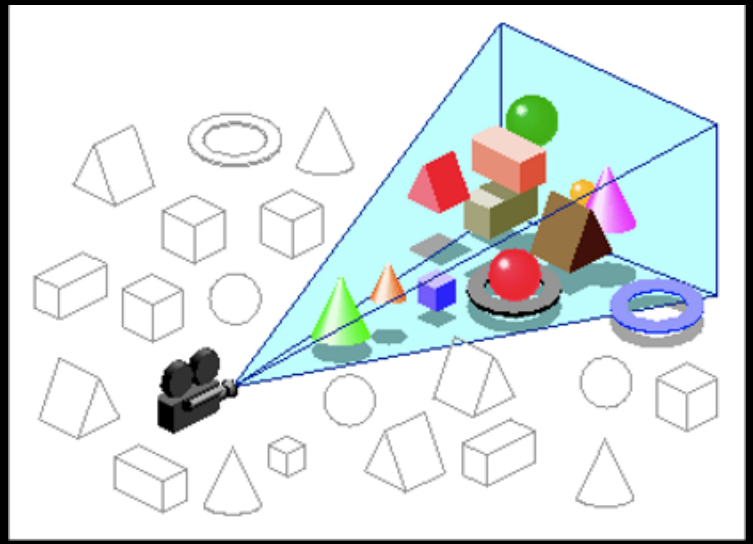
- During onDraw():
  - We tell OpenGL to render every single object
  - Regardless of whether or not it will appear in the scene
- Can we avoid drawing some things?





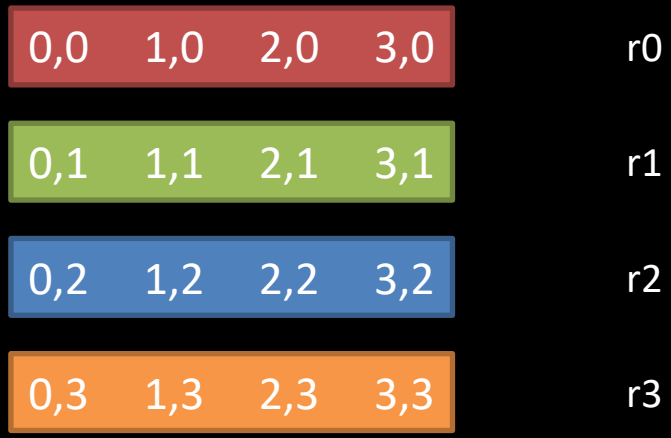
What we should do…

- Instead of telling OpenGL to draw everything, why don't we avoid sending what we know won't be drawn?
- What doesn't need to be drawn?
  - Anything not in the view frustum!
- Only good if we can do this faster than it would take for OpenGL to draw everything



Extracting the View Frustum

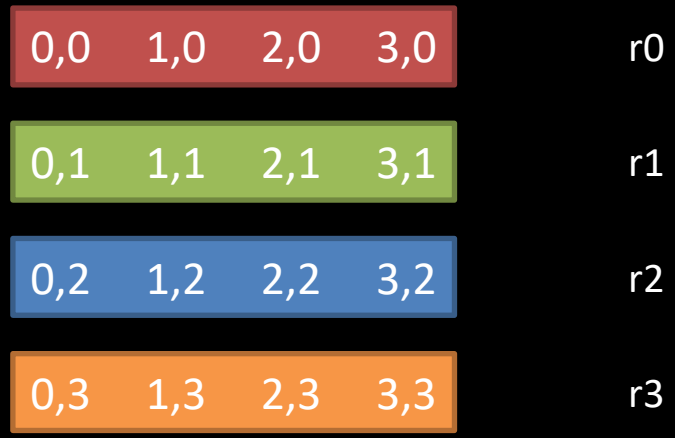
- Frustum is defined by 6 planes
- Planes can be derived directly from the rows of our camera matrices
  - You can get this using
    - camera->getProjection() * camera->getView()
  - Gives us a glm matrix
- Be careful
  - glm uses column-major order, so use the coordinates given here to access the given cells / rows



Projection matrix • view matrix

Extracting the View Frustum

- Plane equation is given by a 4D vector (a,b,c,d):
  - $ax + by + cz + d = 0$
- The 6 clip planes of the frustum are defined below!

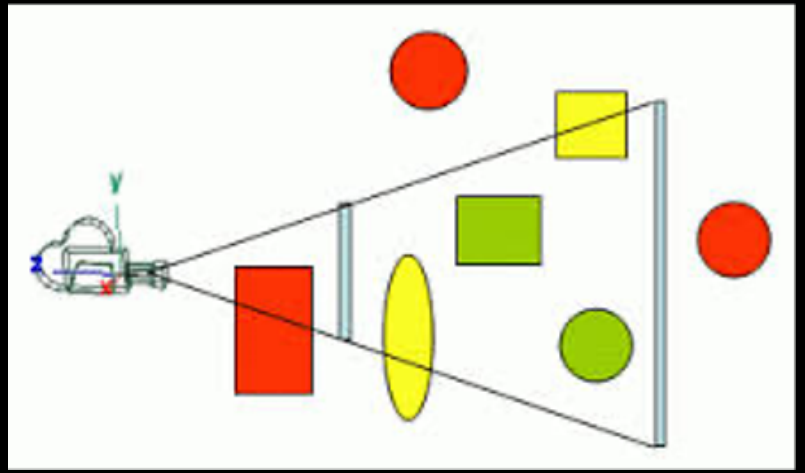


Projection matrix • view matrix

Clipping plane	−x	−y	−z	+x	+y	+z
Plane equation	r3 − r0	r3 − r1	r3 − r2	r3 + r0	r3 + r1	r3 + r2

Frustum Culling Test - General

- Compute 6 plane equations
  - Should be updated whenever the camera changes!
- For each piece of scene geometry:
  - Does the entire shape fall behind one of the planes?
    - Skip rendering, it can't be seen!



Frustum Culling Test - AABB

- AABB (axis-aligned bounding box)
  - Faces parallel to xy, xz, and yz planes
  - Defined by a position and dimensions (convenient!)
- Rejection test: are all 8 corners behind any one plane?
  - For point (x,y,z), behind plane if $ax + by + cz + d < 0$

Frustum Culling Test - Sphere

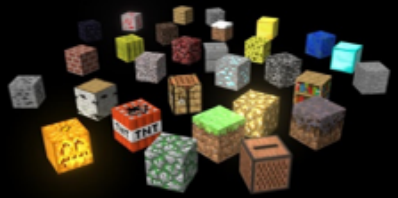
- Sphere
  - Defined by a position and radius (which can just be one of your dimensions!)
- Rejection test: is the center (x,y,z) at least *r* units behind any one plane?
  - If $ax + by + cz + d < -r$
  - Planes must be normalized
    - Divide (a,b,c,d) by $\sqrt{a^2 + b^2 + c^2}$
- You do not have to implement this!
  - Minecraft is all about cubes!
  - May be helpful later!

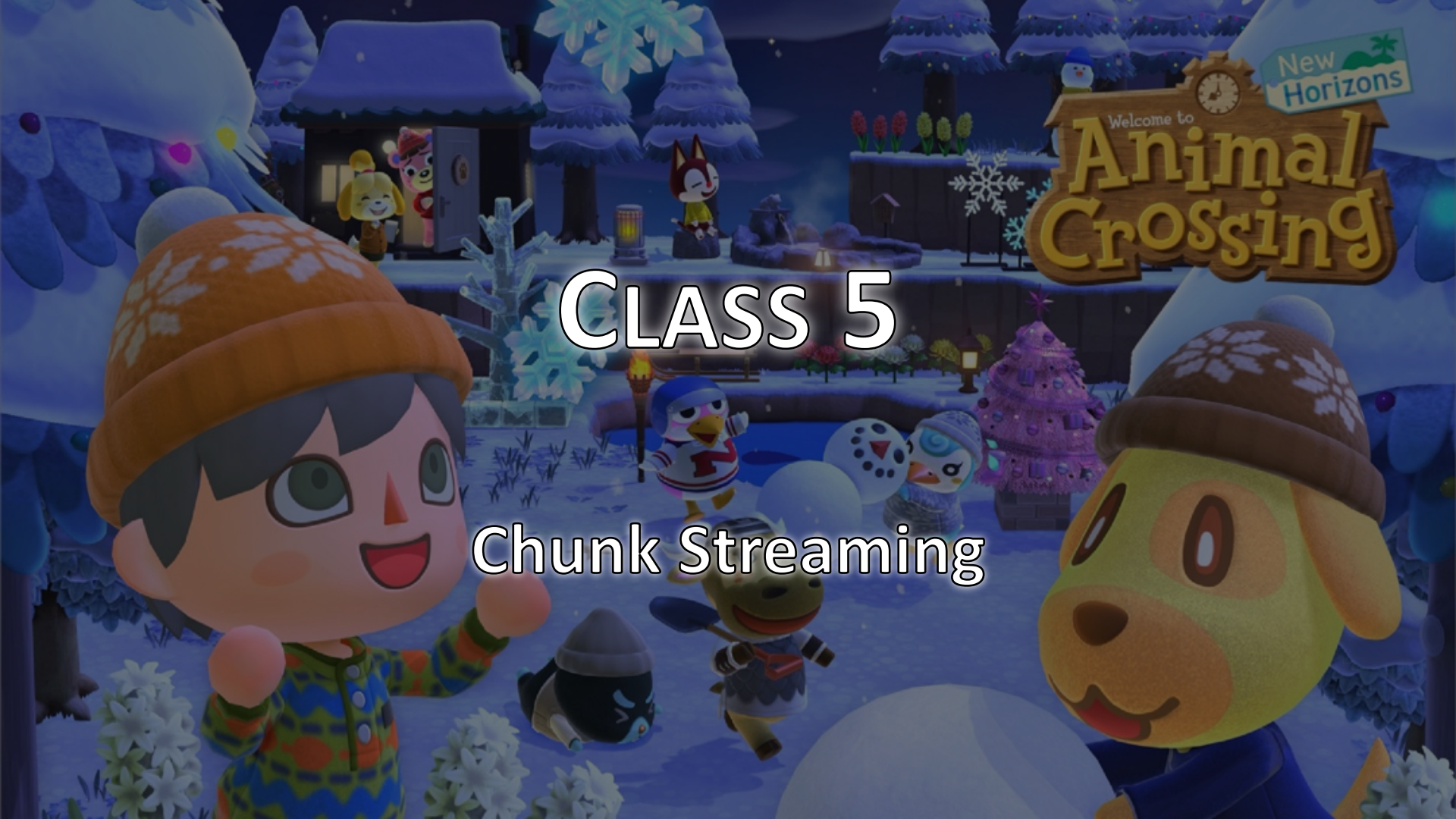
Implementation Notes

- Storing the r vectors
  - In the camera?
    - Only re-compute when camera changes
  - Somewhere else?
    - Up to you
- Design decisions – yay!

Implementation Notes

- What should we cull?
- Individual blocks?
  - Fine-grained
  - Faster to just draw everything
- Whole chunks?
  - Coarse-grained
  - Far fewer culling tests
- Non-environment entities?
  - Depends on # vertices
  - If we have AABB's for them, depends on how many





Class 5

Chunk Streaming

Memory troubles

- 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…

What should we forget?

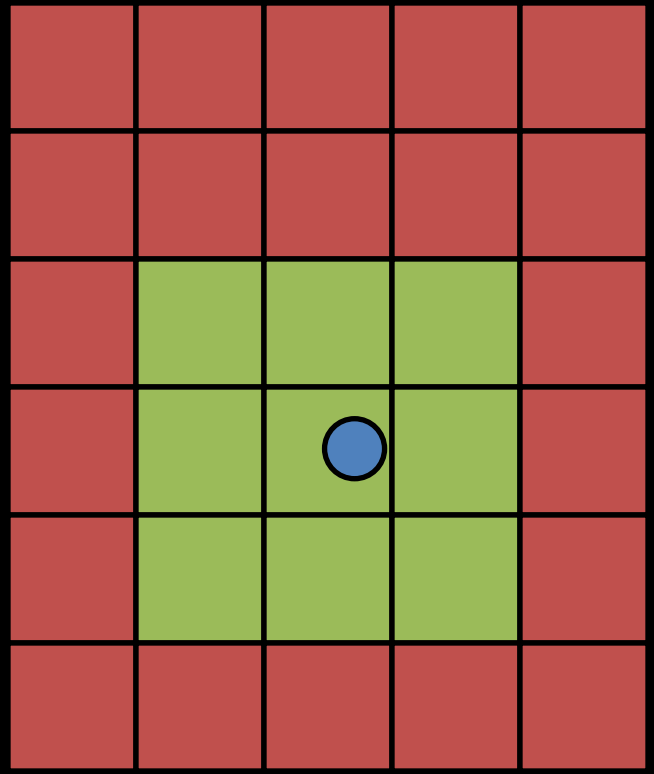
- Solution: store only in memory what the player is likely to interact with
- Two parts:
  - Load chunks only as the player approaches them
  - Unload chunks when the player has moved significantly far away from them

Dynamic World Loading

CHUNK STREAMING

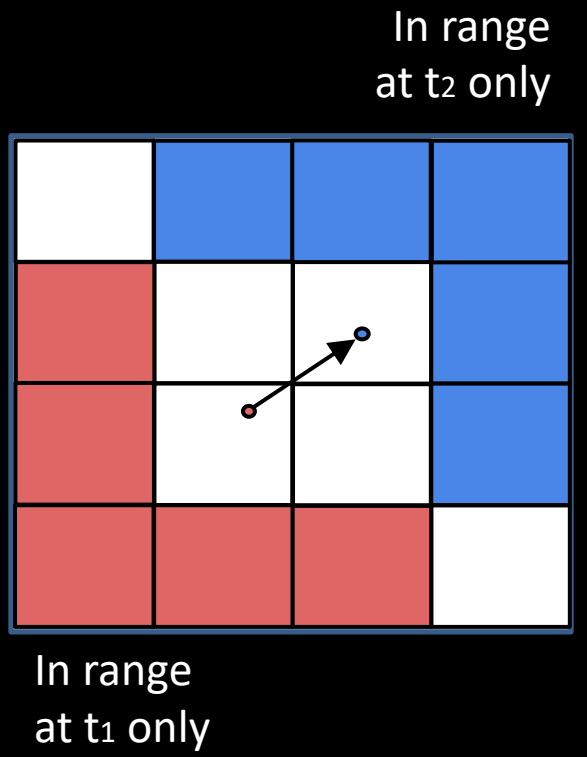
Chunk streaming

- 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



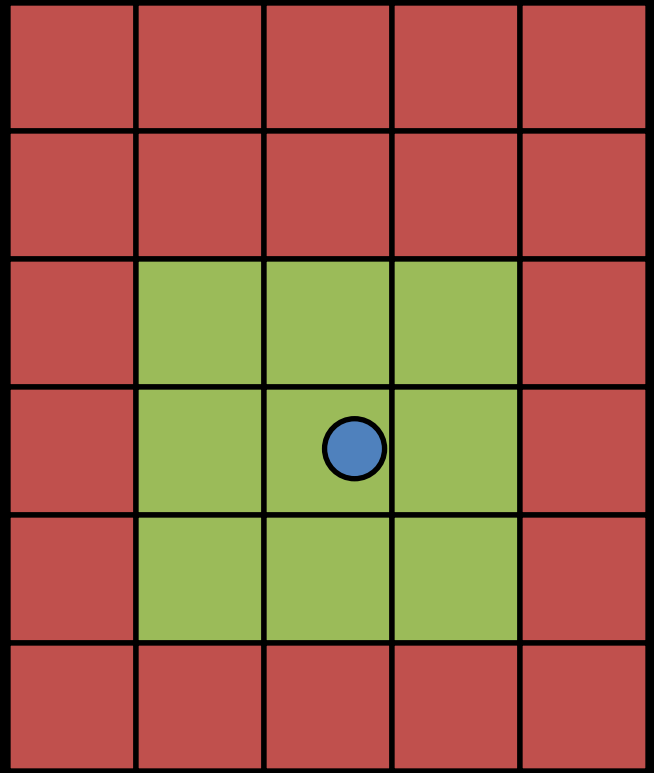
Chunk streaming

- 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 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?
  - Design decisions

Class 5

C++ Tips

Static

- Very helpful C/C++ keyword
- Behavior of static can vary a lot depending on use
  - Member variable vs namespace variable vs local variable causes different results

More Static

- Static variables exist for the "lifetime" of the *translation unit that it's defined in*
  - Translation unit: simplest unit of compilation
  - Consists of the contents of a single source file, plus the contents of any header files directly or indirectly included
- Result: static variables have a single instance (usually)

Namespace Static

- If variable is outside of any functions or class, it can't be accessed from any other translation unit
  - This is known as "internal linkage"
- NEVER do this in headers
  - you end up with a separate variable in each translation unit (i.e., every time you include the header)

```
static int x;


class Foo {

  public:

      void bar();

};
```

Member Variables and static

- A static member variable is shared between all instances of the class
- Important: you need to both declare and define static member variables

Bad

```
//in .h file
class Foo {
public:
    static int x;
    void bar();
};

//in .cpp file
void Foo::bar(){
    Foo::x = 10;
};
```

Good

```
//in .h file
class Foo {
public:
    static int x;
    void bar();
};

//in .cpp file
int Foo::x = 0;

void Foo::bar(){
    Foo::x = 10;
};
```

Static functions

- Less confusing
  - Same as in Java
- A static function can be called without an instance of a class
- Cannot access non-static member variables

```
class Foo
{
private:
    static int value;
public:
    static int getValue()
    {
        return value;
    }
};
```

Const

- Another very helpful keyword
- Declares a constant
  - `int x = 4;  //normal int`
  - `const int y = x;  //value of y cannot be changed`
  - `const int* foo = &y;  //can’t change value of y`
  - `int *const bar = &x;  //can’t reassign bar`
  - `const int *const yum = foo; //can’t change anything`

Const++

- Const member functions
  - void Entity::draw() const { … // can't change Entity
  - Very, very good style
- Using const references in functions
  - void Entity::accelerate(const vec3 &acc) { …
  - Only const functions can be called on const references
  - Much cheaper than accelerate(vec3 acc)

C++ Templates!

- C++ templates are an extremely powerful tool for generic programming
- Allows you to reuse the same code without losing any type specificity
- Can be tricky to figure out though—it's okay if you get lost!

Without templates

```
#include <iostream>
using namespace std;

int square (int x)
{
  return x * x;
};

float square (float x)
{
  return x * x;
};

double square (double x)
{
  return x * x;
};
```

```
main()
{
   int    i, ii;
   float  x, xx;
   double y, yy;

   i = 2;
   x = 2.2;
   y = 2.2;

   ii = square(i);
   xx = square(x);
   yy = square(y);
}
```

With templates

```
template <class T>
inline T square(T x)
{
   T result;
   result = x * x;
   return result;
};
```

```
main()
{
   int    i, ii;
   float  x, xx;
   double y, yy;

   i = 2;
   x = 2.2;
   y = 2.2;

   ii = square<int>(i);
   xx = square<float>(x);
   yy = square<double>(y);
}
```

Template Classes

- In addition to have a templated function, we can have a entire templated *class*
- This is how things like vectors, maps, and the like are implemented

```
#include <iostream>
using namespace std;

template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}
```

What are Templates Good For?

- Use Case #1:
  - Generic methods for adding / getting / removing components!

```
class GameObject {

template <class T>
void addComponent(T std::shared_ptr<T> comp)
{
    …
};

}
```

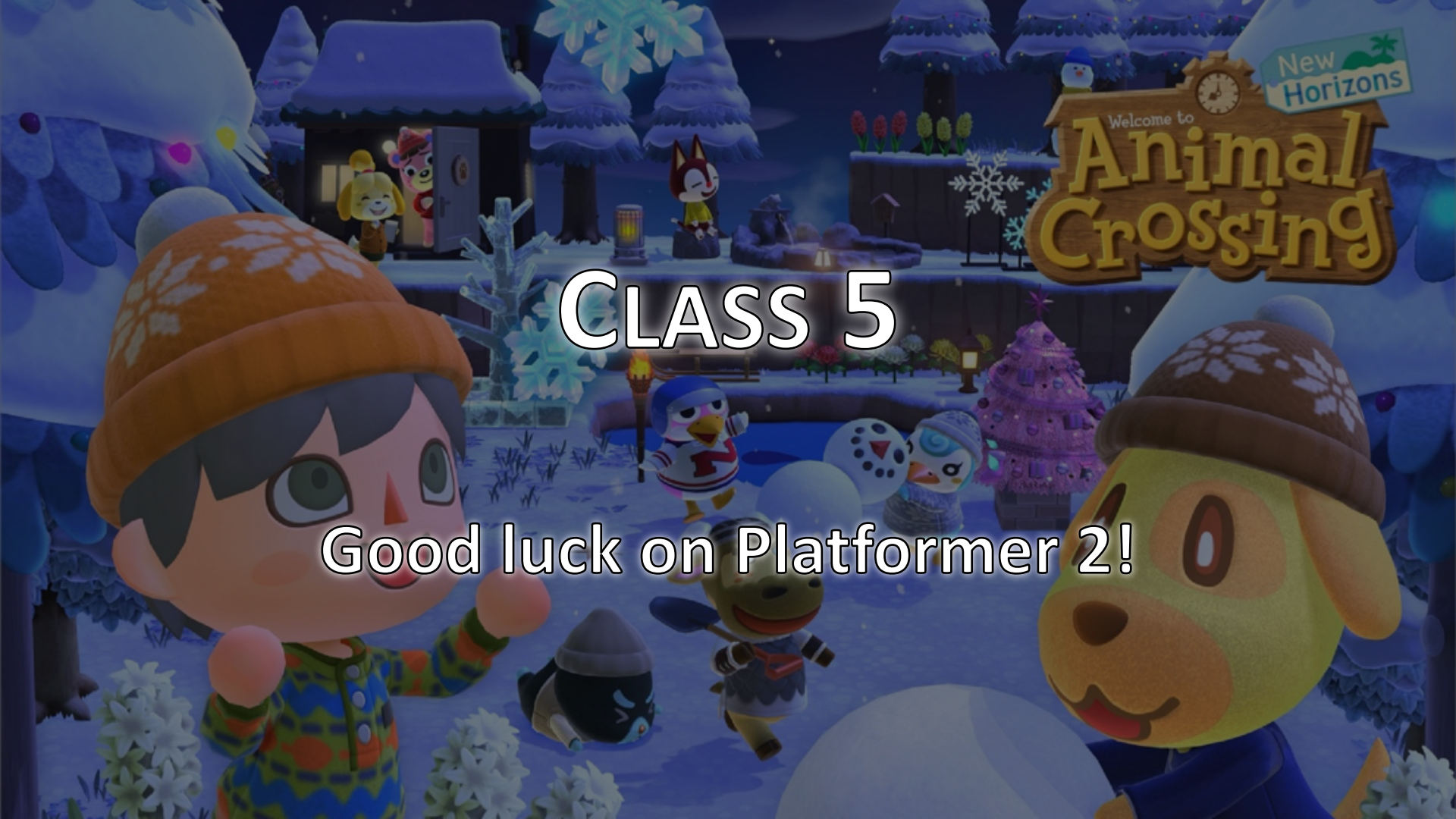
```
class GameObject {

template <class T>
std::shared_ptr<T> getComponent(…)
{
    …
};

}
```

What are Templates Good For?

- Use Case #2:
  - Generic methods for adding / getting / removing systems!

```
class GameWorld {

template <class T>
void addSystem(T std::shared_ptr<T> sys)
{
    …
};

}
```

```
class GameWorld {

template <class T>
std::shared_ptr<T> getSystem(…)
{
    …
};

}
```



Class 5

Good luck on Platformer 2!