
Class 8
Final Project Topics

Final Project Topics

- We will try to cover the “big” features first
- These are not intended to give you the full implementation details
  - Think “high-level overview”
  - You *will* have to do your own research
  - You’re focusing on these 1 or 2 things for multiple weeks, so make sure you’re interested in them
- Don’t wait for us to teach you – start your research now!
  - The staff has pretty good coverage, but even we don’t know some of this stuff
  - Teach us!


Class 8
Networking

Networking

NETWORKING STRATEGIES

The Illusion

- All players are playing in real-time on the same machine
- But of course this isn't possible
- We need to emulate this as much as possible

The Illusion

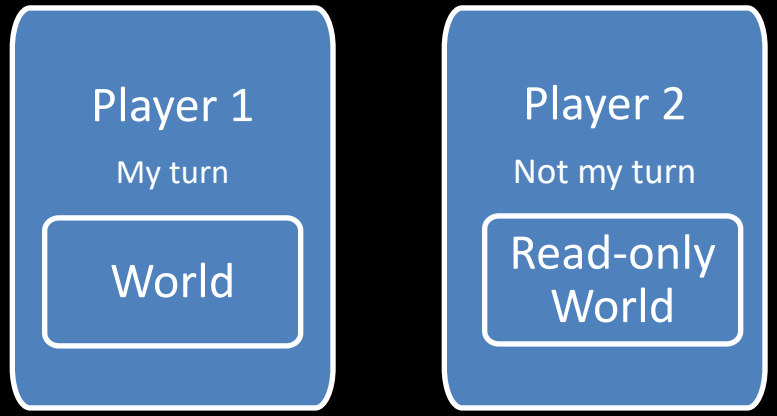
- What the player should see:
  - Consistent game state
  - Responsive controls
  - Difficult to cheat
- Things working against us:
  - Game state > bandwidth
  - Variable or high latency
  - Antagonistic users

Send the Entire World!

- Players take turns modifying the game world and pass it back and forth
- Works alright for turn-based games
- …but usually it's bad
  - RTS: there are a million units
  - FPS: there are a million players
  - Fighter: timing is crucial

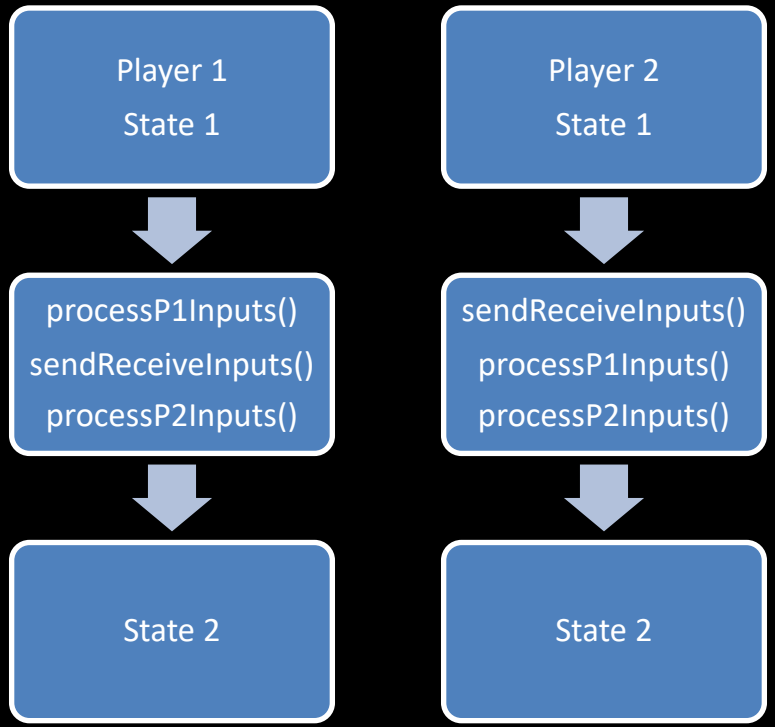
Modeling the World

- If we're sending everything, we're modeling the world as a uniform chunk
  - But it really isn't!
  - Composed of entities, only some of which need input from a player
- We need a better model to solve these problems



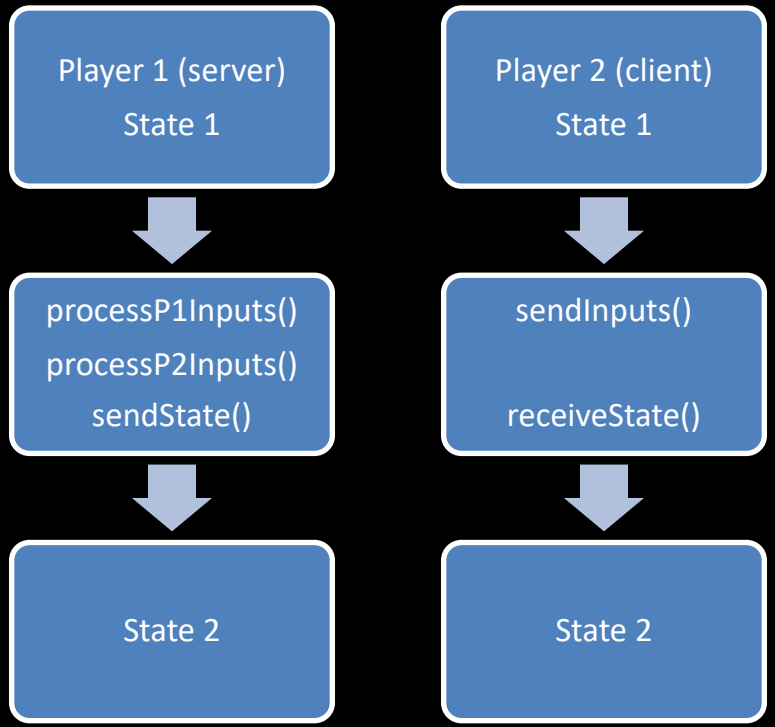
Send Commands

- Model the world as local and shared data
  - Share player information, powerups, etc
  - Don't need to share static level data
- Each player sends the other all actions that alter shared game world
- "Deterministic P2P Lockstep"
- Problem: everything must evaluate the same
  - Or else there are desyncs
- Problem: have to wait for all the other players' commands
  - So everyone is limited by laggiest player



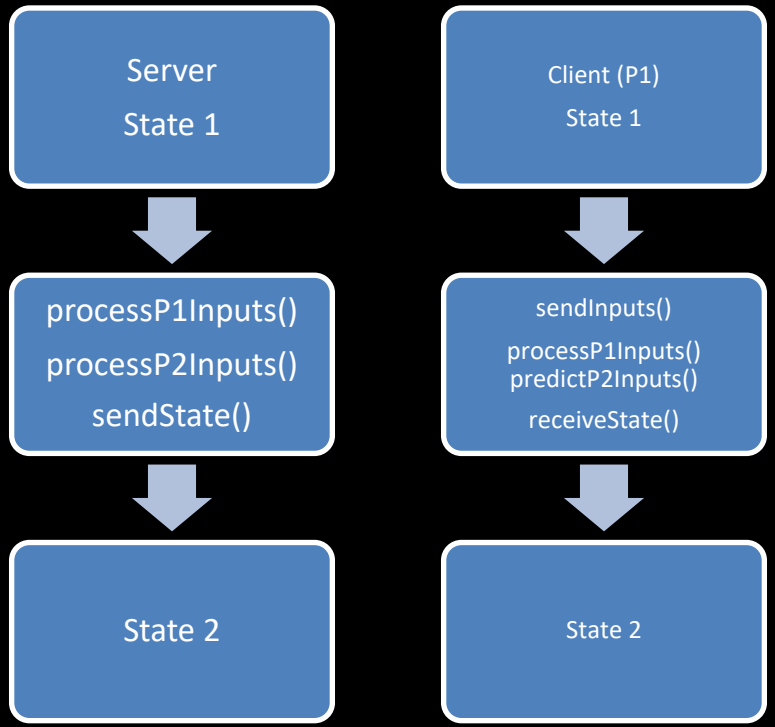
Client-Server Model

- One player is the authoritative server
  - Now we don't have to wait for slow players, just the server
- Other player is a "dumb terminal"
  - Sends all input to server
  - Server updates the world and sends it back
- Problem: client has to wait for server to respond to perform even basic actions



Client-side Prediction & Rollback

- Client responds to player input immediately and predicts new game state
  - This might including predicting other player's inputs
- When the server sends back the authoritative game state, incorrectly predicted client state is overwritten



Client-side Prediction & Rollback

- But we just received a state from the server that was 100ms in the past!
- We can't just replace our local state or else our local state would also be 100ms in the past
  - We lose player input that happened in those 100ms
- Client has to roll back the world and re-apply input that happened since the last known good state
  - This includes predicting new inputs for other players

What about the server?

- Without rollback:
  - In an FPS, would need to lead shots because the server won't register shot until after delay
- With rollback:
  - You could be shot after you think you've taken cover

Masking the Timewarp

- Problem: laggy players re-apply lots of inputs during rollback
- Solution: if the server usually sends states from 100ms ago, apply inputs 100ms late on the client
  - Still send inputs to the server immediately
- Turns a jumpy experience into a smooth, only slightly slow one
  - Very useful if relative timing of commands is important

Edge Cases

- What if…
  - The client disconnects
  - The server dies
  - The client goes insane and sends gibberish
  - The client loses internet for 30 seconds
  - The client is malicious
  - The client changes IP address
- Handling errors well is vital to player experience

Elegant Disconnects

- Handle and respond to IO exceptions
  - Don't just dump a stack trace
- Display informative status messages
- Send heartbeat packets every few seconds
  - Then respond if server/client hasn't received a heartbeat in a while
- Never let the game continue to run in an unrecoverable state!

Resources

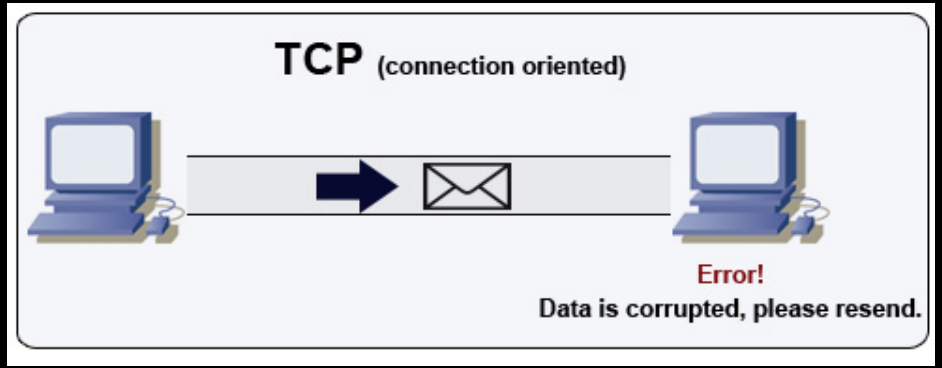
- For a more in-depth discussion about networking models and the concepts of client-side prediction and rollback
  - See this link
- Lots of other resources online

Networking

IMPLEMENTATION

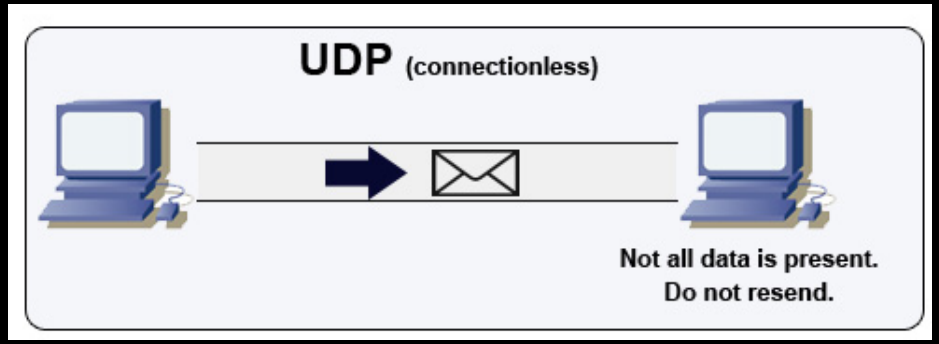
TCP: Transmission Control Protocol

- Abstracts over IP
- All packets are guaranteed to be received and in the correct order
- Good for sending important, permanent data (websites, databases, etc)



UDP: User Datagram Protocol

- A very thin shell around IP
- Much faster than TCP, but no guarantees about reception or order
- Good for information where only the most recent state matters (streaming, etc)



TCP vs UDP

- (Very) generally: action games use UDP and turn-based games use TCP
  - World state updates can be lost without worry, commands not so much
- Can potentially combine them
  - TCP sends important data, UDP sends timely data
- Best choice varies by project
  - (for naïve version, TCP is fine)

C sockets

- So much more than we want to cover in class
- Pros:
  - Full control over network throughput
  - Worth 5+ points
  - You will learn a lot
- Cons:
  - Oh so much more complicated
  - Will require multithreading, synchronization, and an incredibly well thought out design

- Start with CS033's snowcast project
  - http://cs.brown.edu/courses/csci1972/handout/snowcast.pdf

C++ QSockets

- Qt has QTcpSocket and QUdpSocket!
- Pros:
  - Far easier to set up than standard sockets
  - Convenient blocking, non-blocking IO calls
- Cons:
  - Still sending/reading bytes
  - Still need multithreading, synchronization, and a good design
  - Ton of error checking required

- Better, but still not perfect. So…?

Networking

RAKNET

The RakNet library

- Open-source games networking library
  - Recently bought by Oculus!
  - Plugin-style
- Used by some *really* legit engines
  - Unity, Havok, Minecraft
- Find it here: http://www.jenkinssoftware.com/

The basics

- Basic client-server or P2P connections
  - Read and write threads made for you!
- BitStreams that can serialize:
  - Primitives (char, int, long, etc…)
  - !!! Structs !!!
- Basic packet objects with metadata

Isn't that everything…?

- Now sending data is easy…great!
- Still have to…
  - Pick what to send
  - Pick when to send
  - Interpret what is sent
- What if I have 1000 entities in my world?
  - Entire world may be too much data…
  - Need some complex ID system
- Gee, it would be great if…

RakNet does that too!

- Introducing ReplicaManager3!
  - Networked entities inherit from Replica3
  - RakNet gives you callbacks for all serialization events
    - onConstruct
    - onDestruct
    - onSerialize (each tick)
- *A LOT* of setup required, but works amazingly well
- Probably better to extend it a bit for simpler callbacks

Using ReplicaManager3

- Have entities override some “NetworkedEntity” class that does most of the setup
  - Most of it is the same for every object
- Determine where entities are made and destroyed
  - Client or server side?
- Override serialization methods
  - Feed stuff in/out of a BitStream in the right order!
- Register created/destroyed entities with RakNet
- ???
- Profit!

Sounds great!

- Since you decide what's serialized, you can avoid sending things other clients don't care about…
  - But what about things that don't change often?
- VariableDeltaSerializer!
  - Only sends data that hasn't changed since last tick!
  - More work now, less network throughput
- Space-time tradeoff probably worth it…

I’m Sold!

- Lobby system
- “Fully connected mesh” host determination system
- Authentication protocols
- Team management
- !!! Voice chat !!!
- SQLite3 databases
- And so much more…

In conclusion…

- RakNet is a beast
- Pros:
  - Handles the nitty-gritty threads/sockets details for you
  - RM3 really simplifies the design process
  - Lets you focus more on engine design
  - Fast.
- Cons:
  - A lot of setup required for RM3
  - You won't learn as much ☹

Networking - RakNet

QUESTIONS?


Class 8
Advanced Graphics

Advanced Graphics

PARTICLES

Particles

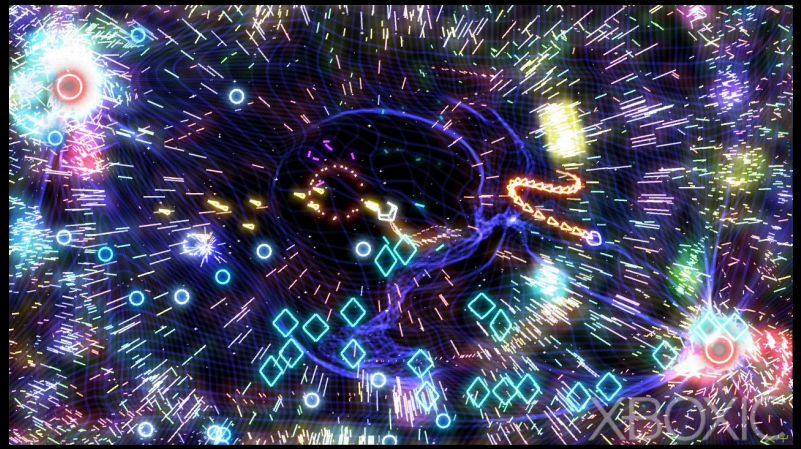
- What is a particle?
  - A particle is a tiny entity that is used in massive quantities to create a visually pleasing effect
  - Used to model effects like fire, liquids, hair, etc.
  - Conceptually old—first paper published in 1983

Particles

- What makes particles look good?
- Fade out or get smaller linearly over their lifespan
  - Once the particle is completely gone or transparent, it can be removed from the world
- Adding some kind of randomness to how they move
  - Starting position, velocity, acceleration, color, size, shape

Particles

- Particles are great
- But they are very slow if not done correctly
- Things that make them slow:
  - It's a lot of information to tick
  - It's a lot of information to draw
  - There are way too many of them to consider doing collision detection against each other



Particles

- What shape should my particles be?
- Sphere?
  - Is limited if you want transparent particles
  - Too many vertices
- Quad!
  - Use a texture like this one
  - Rotate the quad to always face the camera
  - This texture has a black background and no alpha information
    - Use graphics->enableBlendTest(Graphics::BLEND_FUNC::ADD)
    - This says take all of the background, and add the color of this particle on top of it
    - Particles that are denser will appear brighter

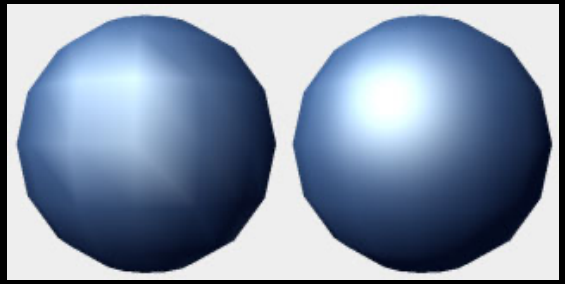
Particle Optimizations

- Reduce the amount of information in your particles
  - Vector3 position, Vector3 velocity
  - maybe some noise values to make them scatter
  - Less information to tick
- Don't make your particles GameObjects
  - Keep them in a separate list so that you can tick and draw them all at once
  - Binding the particle texture once and then drawing all your particles without unbinding and rebinding the texture is a HUGE improvement
- Use GL_TRIANGLE_STRIP instead of GL_TRIANGLES triangle strips to draw
  - Less data needs to be sent to the GPU with triangle strips
  - A particle is just a quad, so you only have 4 vertices vs 6 vertices
- Don't collide them with each other
  - If you must have your particles collide, have them collide only with entities or terrain, not with each other
  - This means they also don't need a shape, so they take up less space
- Keep them in an old-fashioned C-style array
  - This limits the number of particles you can have (which is probably a good thing)
  - Keeps all the memory contiguous
  - Once you are trying to allocate more particles than you have room for, the oldest ones are kicked out first
- Tick them in your draw loop
  - Only having to iterate over them once
- Do them on the GPU (see CS 1230 lab)

Advanced Graphics

DEFERRED LIGHTING

Motivation



- Per-vertex lighting is ugly
  - Can increase number of vertices, but this requires lots of extra memory
- Per-pixel lighting looks a lot better
  - But is much slower than per-vertex, must calculate lighting equation for each **pixel** instead of each **vertex**.
  - Naïve implementation has many wasted calculations
    - Calculates lighting on pixels that are later overwritten by a closer triangle
- Deferred lighting removes most of the wasted calculations and provides further optimizations
- Deferred lighting is an optimization, not a lighting model. You still have to choose a lighting model (for example Phong Lighting)

Overview

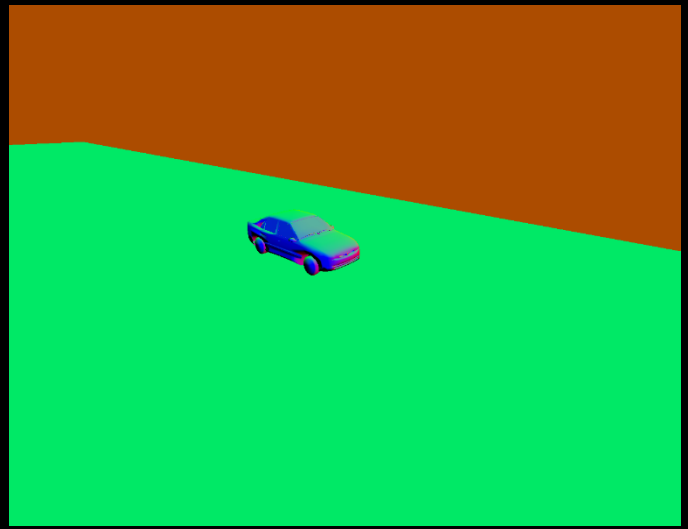
- How can we avoid wasted calculations?
  - Only calculate lighting once for each pixel
  - But the fragment shader has no way of knowing if the value it is calculating will be the final value for that pixel
  - Solution: Multiple passes
- First pass
  - Render geometry and keep track of data necessary to calculate lighting
- Second pass
  - Calculate diffuse and specular values that are independent of material
- Third Pass
  - Combine diffuse/specular from second pass with geometry and material (Object color for example) to complete lighting model

Passes

- A “pass” just means generating a texture (or multiple textures).
- Use framebuffer objects (FBOs) to group textures
  - FBOs are basically a collection of textures
  - The FBO allows you to write to these textures (instead of writing to the screen)
  - Default framebuffer is the screen
    - graphics->setDefaultFramebuffer();
  - Look at FBO class methods
    - constructor
    - bind (sets framebuffer as active)
    - getColorAttachment (gets texture for one of the framebuffer’s color attachments)
  - And Graphics methods
    - addFramebuffer, setFramebuffer
- For example:
  - First pass writes to “Texture1” (using “FBO1”)
  - Second pass reads from “Texture1” and writes to “Texture2” (using “FBO2”)
  - Third pass reads from “Texture2” and writes to the screen

First Pass

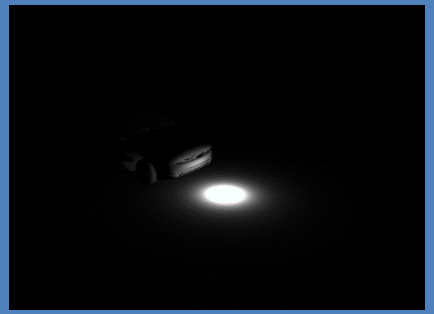
- Takes in all our geometry as input
  - Doesn't need material properties (ex. textures)
  - This just means you need to draw everything in the scene using your first pass shader
- It outputs exactly the information we need to calculate lighting
  - Normals
  - Positions
  - Shininess – store as alpha channel of normal



Second Pass (1/2)

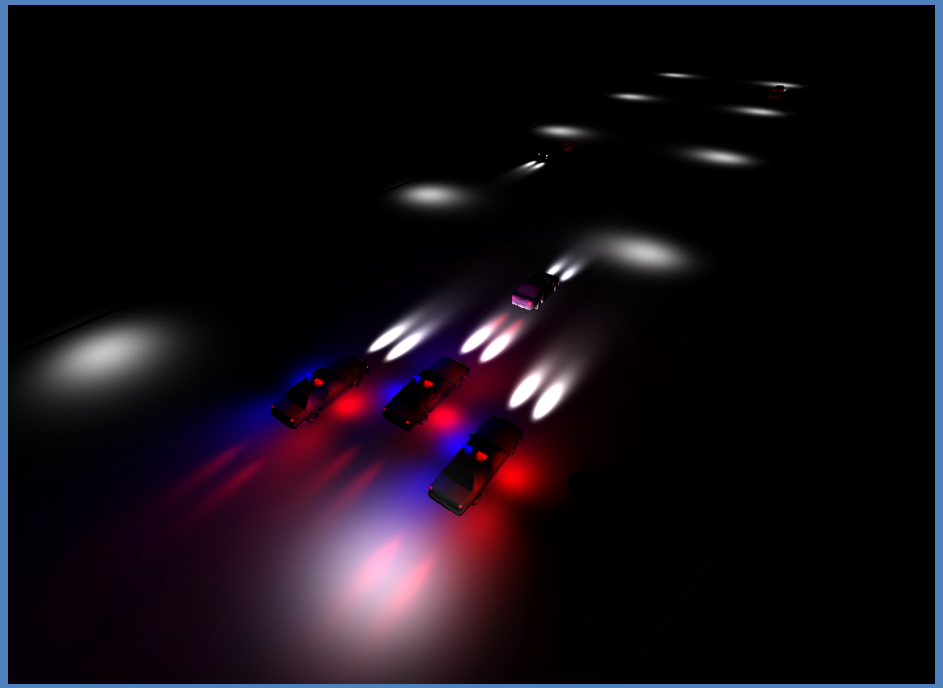
- Takes in normals, shininess and positions from first pass and light data
- Outputs diffuse and specular light contributions
  - Can save space by rendering to a single texture and storing specular contribution as alpha (but we only get monochromatic specular highlights)
  - Or render diffuse and specular to separate textures
- How do we send light data to GPU?
  - For each light:
    - Set necessary uniforms (position, direction, color, etc…)
    - Naïve: render full-screen quad to run the fragment shader on each pixel
      - Can do better, see slide 10
  - But each light would overwrite data from previous light
    - Solution: `graphics->enableBlendTest(Graphics::BLEND_FUNC:ADD)` for additive blending

Diffuse: $(\vec{n} \cdot \vec{l}) * lightColor$

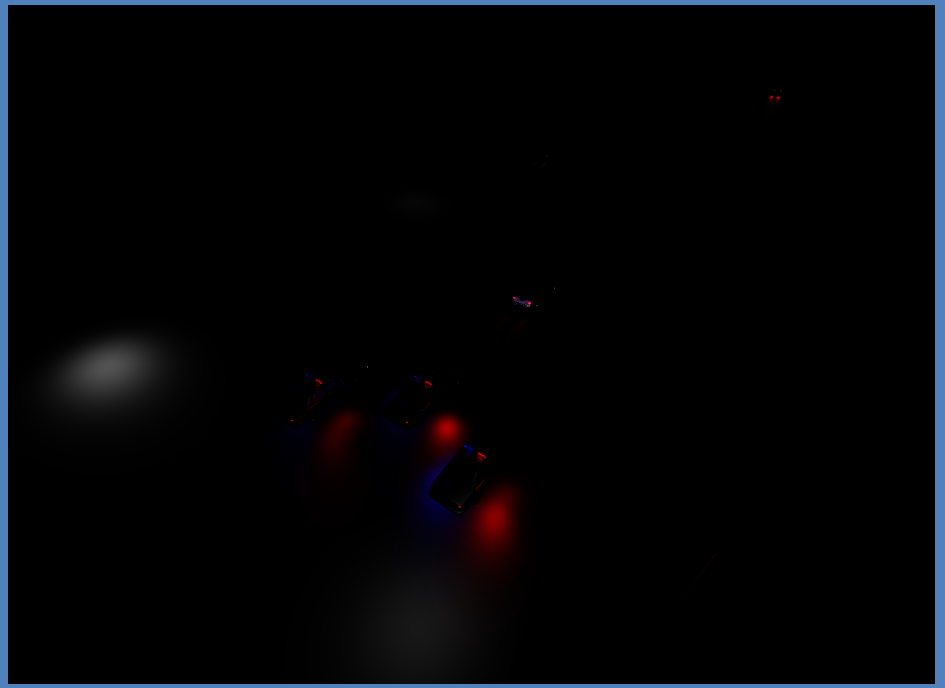


Specular: $(\vec{r} \cdot \vec{v})^n * lightColor$

Second Pass (2/2)



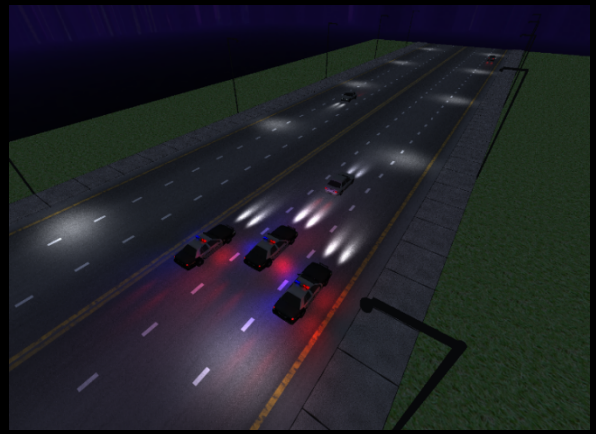
Diffuse

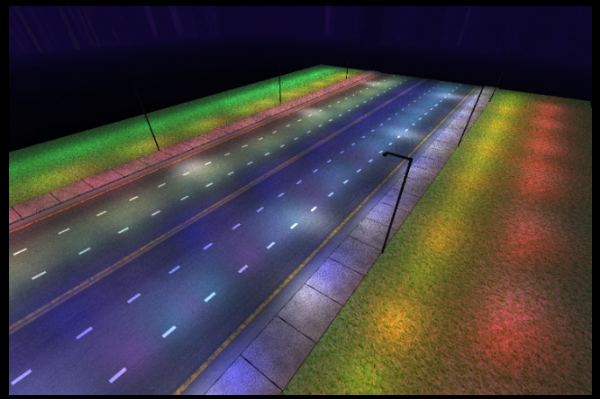


Specular

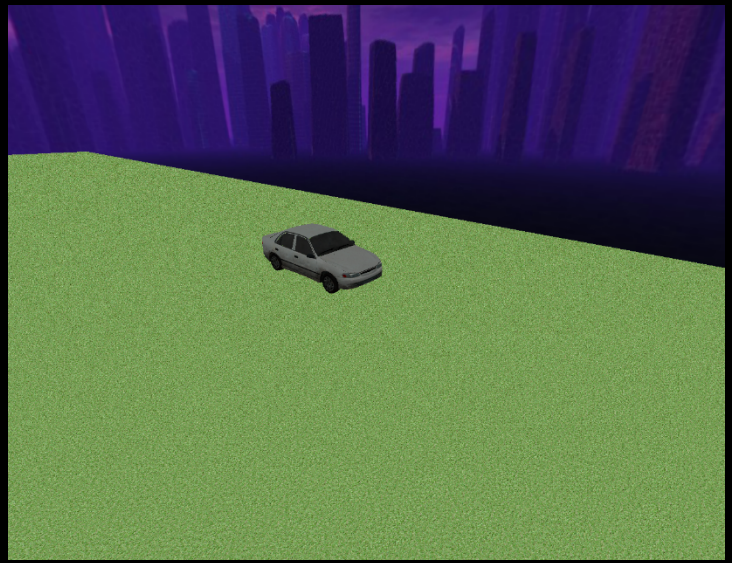
Third Pass (1/2)

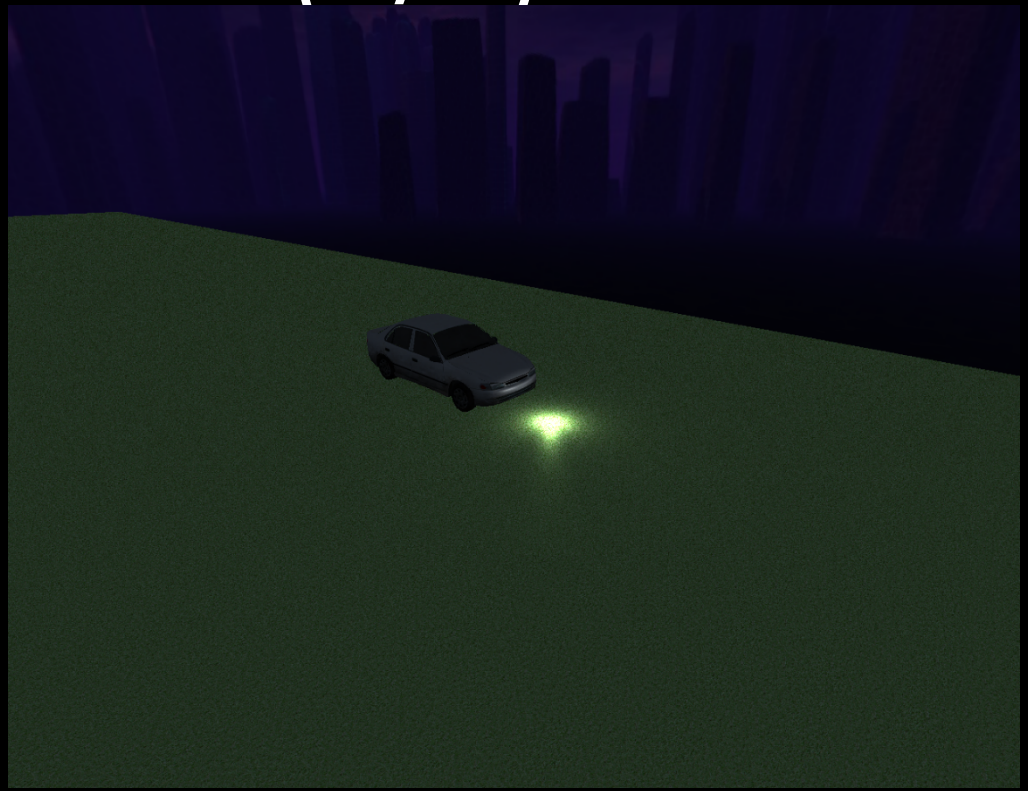
- Takes in diffuse and specular contribution from second pass and geometry, textures, etc… (whatever we need to calculate object's diffuse color)
- Render the scene again, this time applying any materials and finishing the lighting equation (i.e. finish calculating diffuse and specular term and add ambient + diffuse + specular)
- Output is our final lit scene which goes to the screen





Third Pass (2/2)



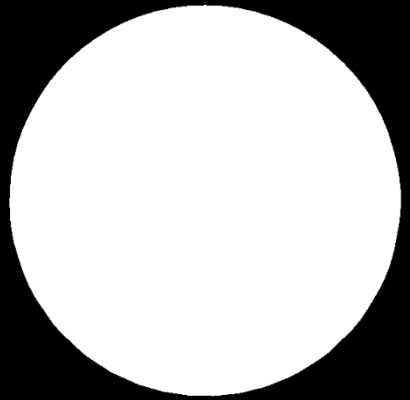


Optimizations

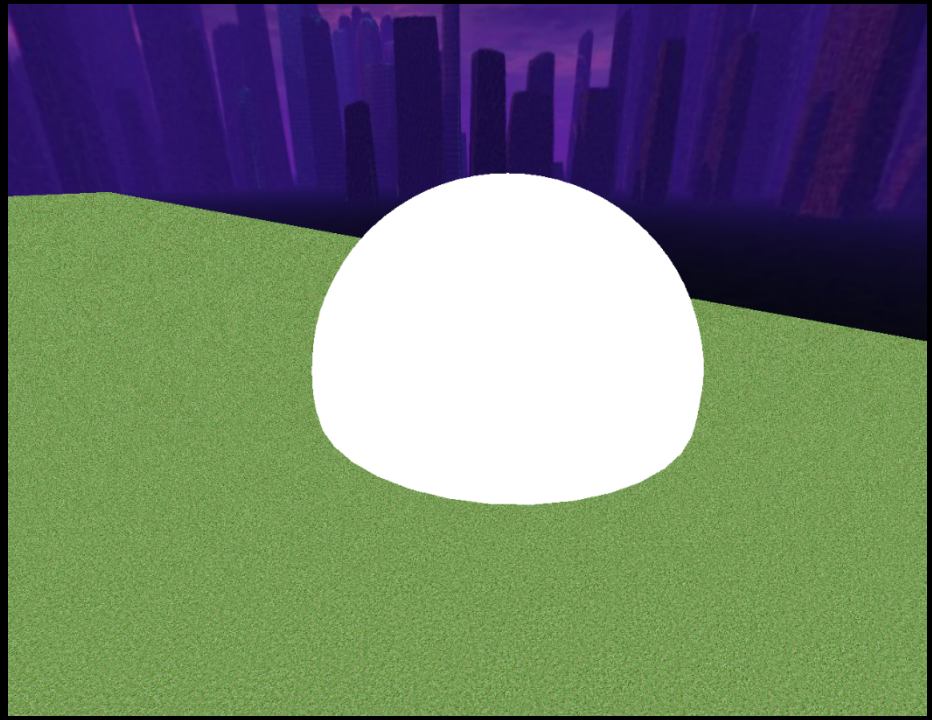
- Instead of calculating lighting on every pixel for every light, calculate lighting on only the subset of pixels that a light can possibly affect.
- Restrict the lighting calculations to the geometric shape that represents the volume of the light. In the second pass render this 3D shape instead of a full-screen quad.
  - Point light: sphere (radius usually based on attenuation)
  - Spot light: cone
  - Directional light: full-screen quad
- What if the camera is inside the shape of the light?
  - Represent light as a full-screen quad
- We will still have some wasted calculations, but this is much better (especially for small lights).

Optimizations

How the light is represented

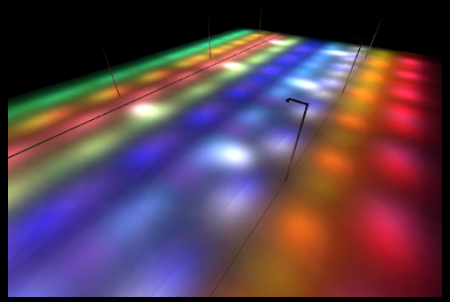


Light visualized in scene

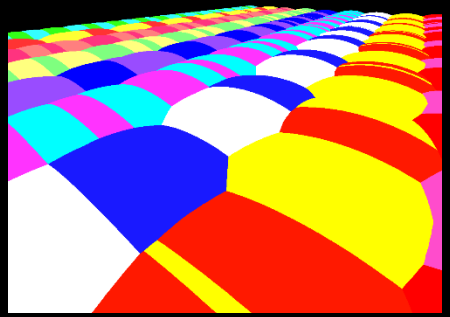


Optimizations

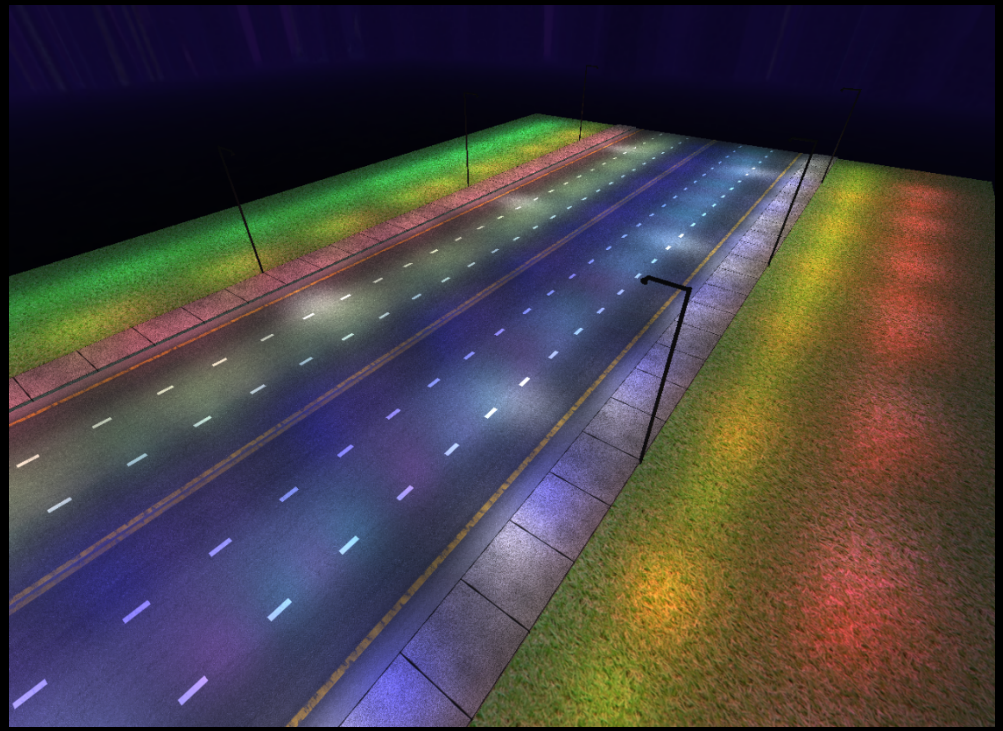
Diffuse contribution



Visualization of lights



Final scene



Optimizations

- The second pass needs to know the position of each pixel in world space
  - Our first pass shader can easily write this position to a texture
- Doing this uses an extra texture (i.e. twice as much memory)
- Instead, can use the depth buffer from the first pass.
  - First pass already uses a depth buffer, so we don't need any additional space.
- Depth buffer has z value from 0 to 1 for each pixel in screen space (convert x/y to screen space based on width/height).
- Use the (x,y,z) triplet in screen space and the inverse projection and view matrices to transform to world space.

Deferred Shading

- Deferred shading is another method for speeding up per-pixel lighting, almost the same as deferred lighting.
  - Note that the word “shading” here doesn’t refer to interpolating lighting values, its just the name of this technique
- Uses only 2 passes
  - First pass renders geometry storing normals as in deferred lighting, but also stores all material properties (diffuse color, for example) in one or more additional textures
  - Second pass calculates lighting and uses material properties to calculate final pixel color
- Pros: Less computation (don’t need to render the scene twice)
- Cons: Uses more memory and bandwidth (passing extra textures around)

Overview

- Deferred Lighting
  1. Render normals, positions, and shininess to textures
  2. Calculate diffuse and specular contributions for every light and output to textures
  3. Render scene again using diffuse/specular light data to calculate final pixel color (according to lighting model)
- Deferred Shading
  1. Render normals, positions, shininess, and material properties to textures
  2. Calculate lighting for every light and combine with material properties to output final pixel color

Disadvantages of Deferred Rendering

- Can't easily handle transparency (this is a generic issue with z-buffer rendering techniques)
- Solutions:
  - Naïve: Sort transparent objects by distance
    - Slow
    - Can't handle intersecting transparent objects
  - Order-independent transparency: Depth peeling
  - Use forward-rendering for transparent objects
    - Forward-rendering is the standard rendering pipeline that you've been using
- Can't use traditional anti-aliasing techniques
  - MSAA (Multisample anti-aliasing), one of the most common AA techniques, doesn't work at all with deferred lighting
  - Usually use some sort of screen-space anti-aliasing instead (FXAA, TSAA, MLAA, SMAA)

Advanced Graphics

VOLUMETRIC EFFECTS

Volumetric Effects

- Volumetric glow (fake scattering)

Volumetric Effects

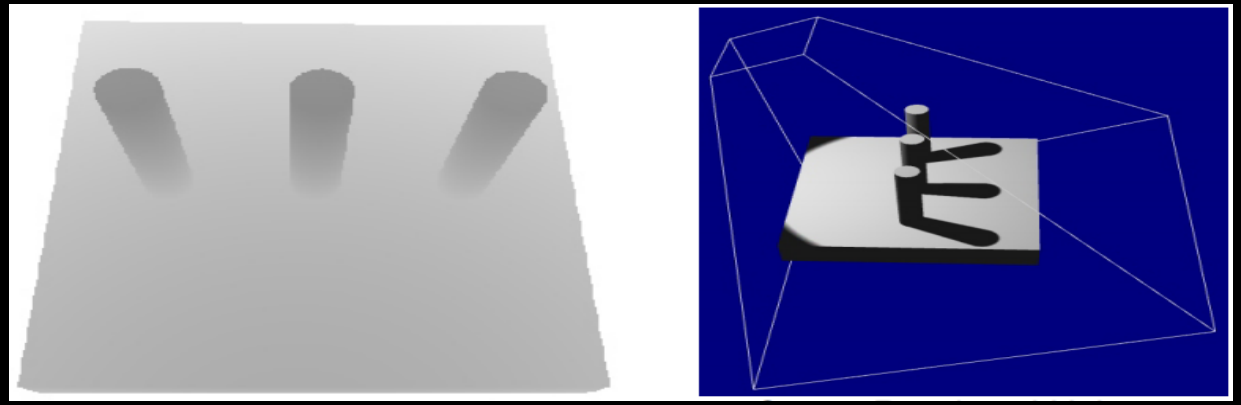
- Volumetric glow (fake scattering)
  - Blend glow color over every pixel
  - Fade off using closest distance from light source to line segment starting from eye and ending at object under pixel
  - Requires deferred shading for position of object
- Rendering to entire screen is expensive
  - Fade off to zero at some radius
  - Only need to draw pixels within that radius in world space, will be cheap for a far away effect
  - Render using inside-out sphere with that radius

Advanced Graphics

SHADOW MAPPING

Shadow Mapping

- Need to test whether a pixel is in shadow
  - Render scene from light's point of view
  - Depth map stores closest point to light
  - Render the scene from the camera, projecting each point back into the light's view frustum
  - Shadow if depth of projected point > depth map

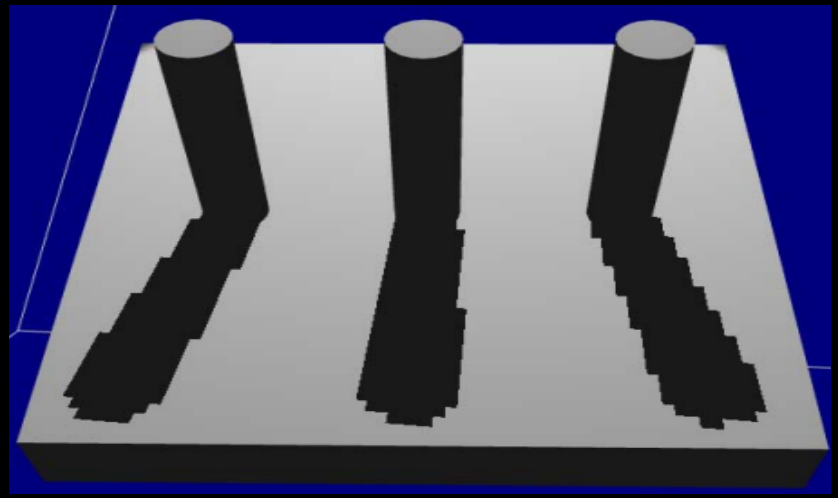


Shadow Mapping

- Need to fit frustum to light
  - Directional light => parallel rays => orthographic
  - Spot light => frustum => perspective
  - Point light => rays in all directions => use cube map

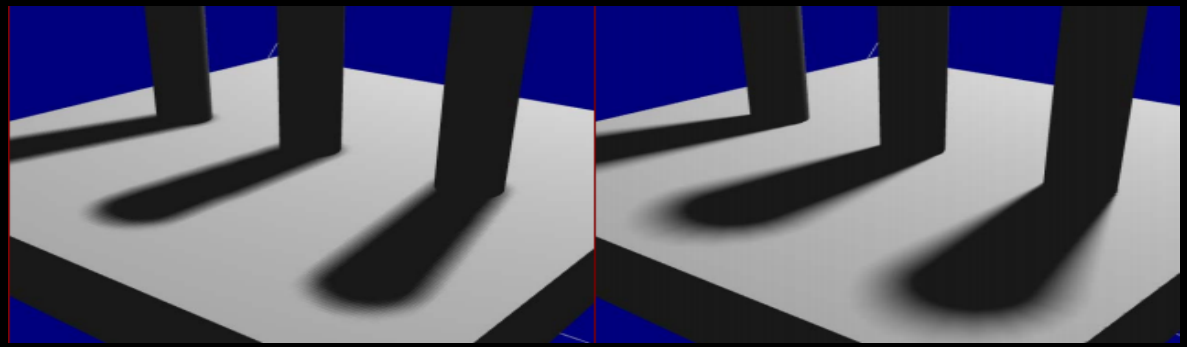
Shadow Mapping

- Problem: Jagged edges
  - Shadow map resolution varies across scene
  - Increasing resolution helps, but uses more memory



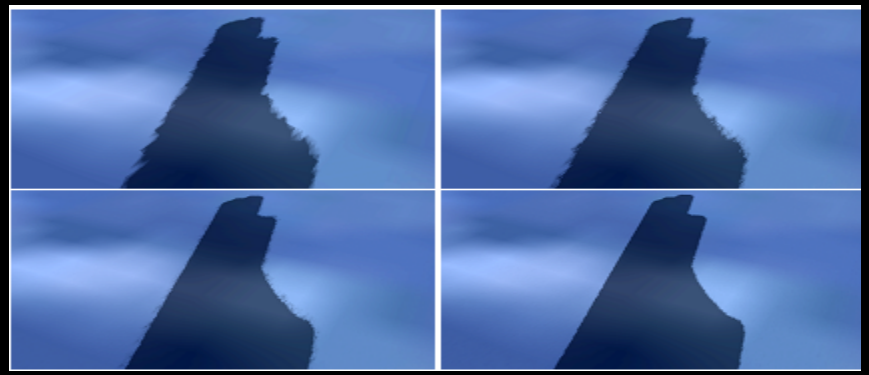
Shadow Mapping

- Fix: blur or fuzz out boundaries
  - Multiple nearby shadow tests are made per pixel and are averaged together
  - Called PCF: Percentage Closer Filtering
    - May use randomized sample patterns
    - May use variable blur size since shadows get more blurry away from caster and area lights



Shadow Mapping

- Fix: Average out over multiple frames
  - Reproject previous frame (if moving camera)
  - Jitter shadow map per frame for more samples
  - Weight by confidence (distance to texel center)
  - http://levelofdetail.wordpress.com/2008/11/06/pixel-correct-shadow-maps-with-temporal-reprojection/



Cascaded Shadow Maps

- Shadow mapping has problems
  - Resolution varies across scene
  - One shadow map per object doesn't scale
- Idea: fit several shadow maps to camera
  - Want uniform shadow map density in screen-space
  - Objects near eye require higher world-space density than objects far away
  - Use a cascade of 4 or 5 shadow maps
  - Each one is for a certain depth range of the scene
- Used in almost all modern games

Cascaded Shadow Maps

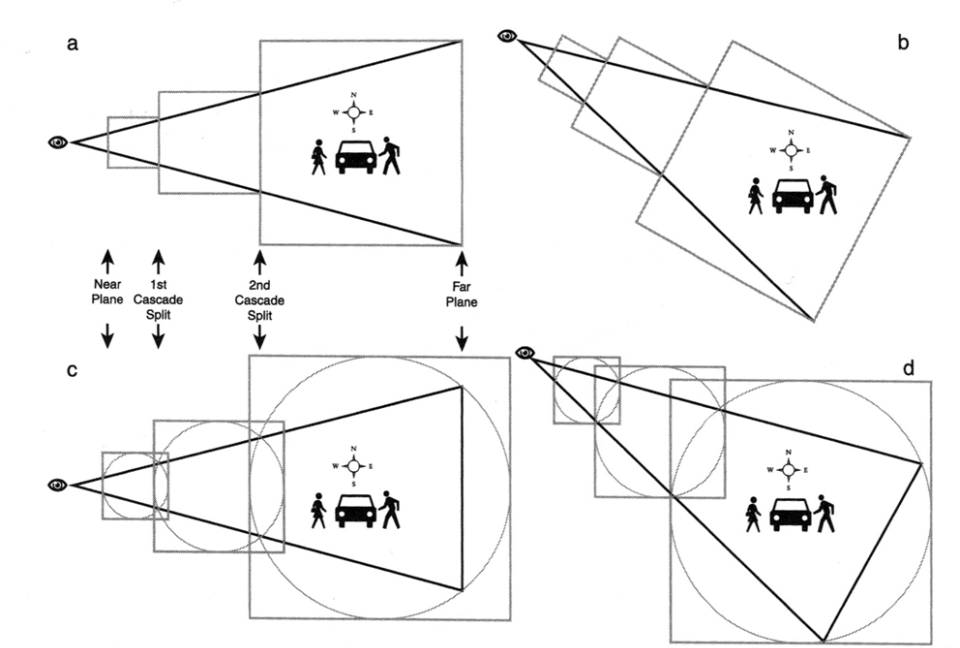
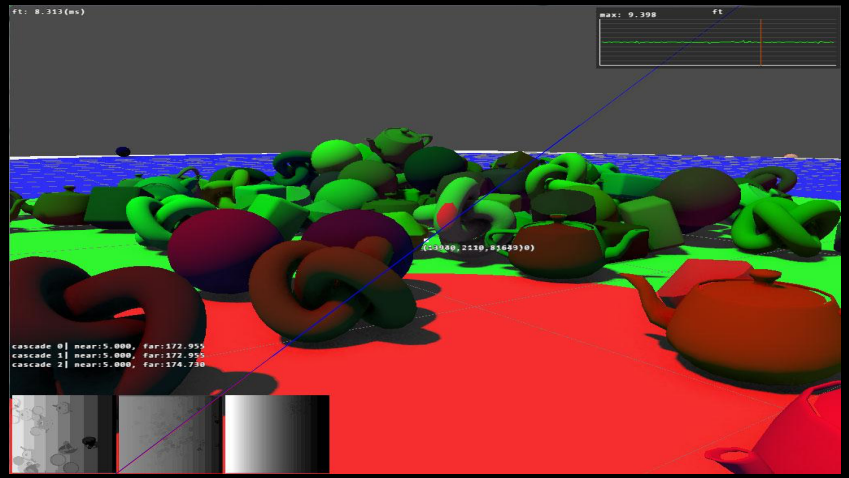


FIGURE 4.1.2 The view frustum in world space split into three cascade frustums and their corresponding shadow map coverage. We use a top view with the light direction pointing straight down the horizontal world plane.

Cascaded Shadow Maps

- Use depth in shader to choose cascade
  - Can blend between two closest cascades to smooth out discontinuities



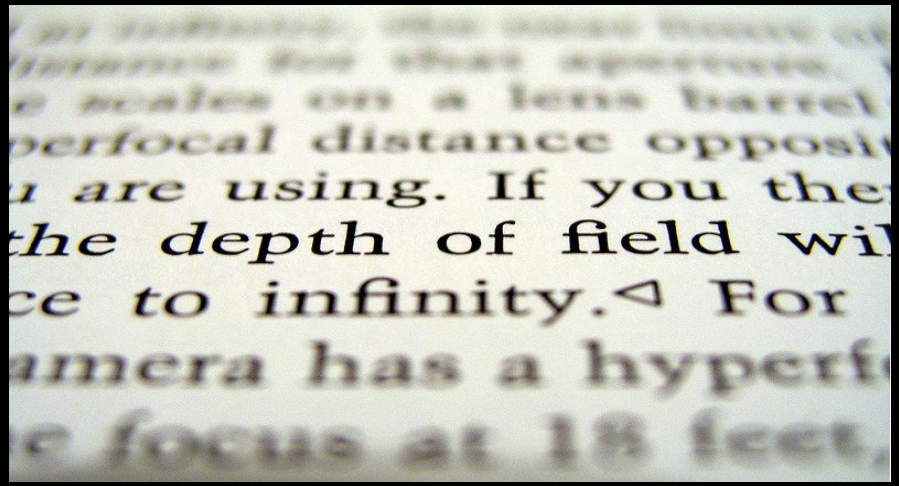
Scene with a cascade of 3

Advanced Graphics

DEPTH OF FIELD

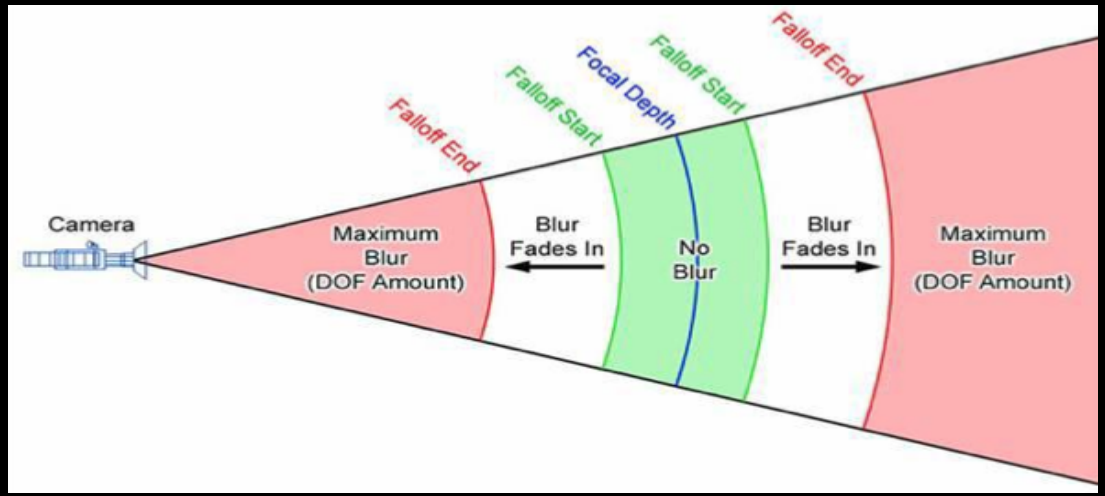
Depth of Field

- Out of focus blur in real cameras
  - Only one depth where objects are in focus
  - Focal blur increases in both directions away from that depth



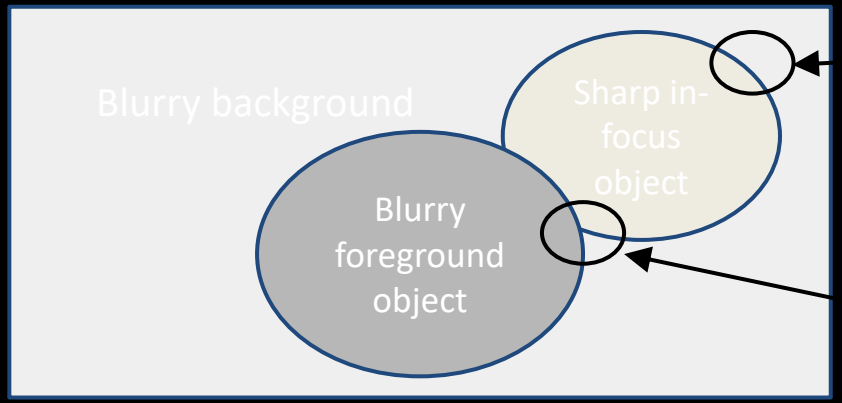
Depth of Field

- Model with post-process blur
  - Vary blur radius based on scene depth: slow
  - Interpolate between 3 images blurred with different radii: fast



Depth of Field

- Discontinuities are problematic (halos)
  - Don't use sharp objects in background blur
  - Blur over sharp objects for foreground objects



Background blur should not use in-focus pixels

Blurry foreground objects should contribute to sharp objects

Depth of Field in Starcraft II

- Avoid sharp halos
  - Buffer of per-pixel blur radius
  - Weigh blur samples by radius buffer at sample point
  - Renormalize to sum to 1 again
- Halos around blurry objects
  - Compute blurred radius buffer
  - Compute blurred depth buffer (local average depth)
  - If average depth < current depth, use radius from blurred buffer, otherwise use radius from sharp buffer

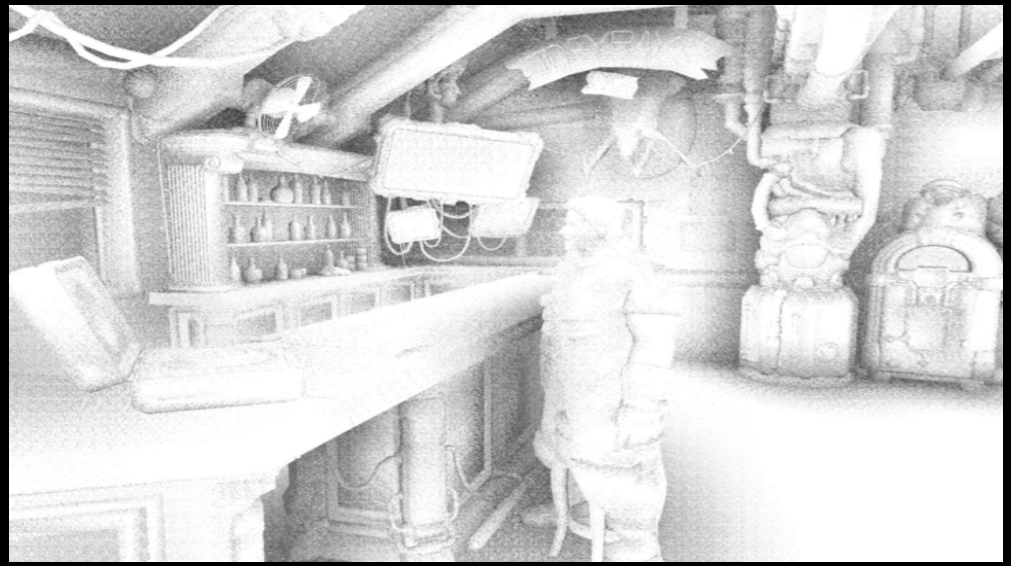
DOF in Starcraft II cutscene

Advanced Graphics

LIGHTING EFFECTS

Screen-Space Ambient Occlusion

- Darken ambient term in occluded areas
  - Approximates indirect lighting

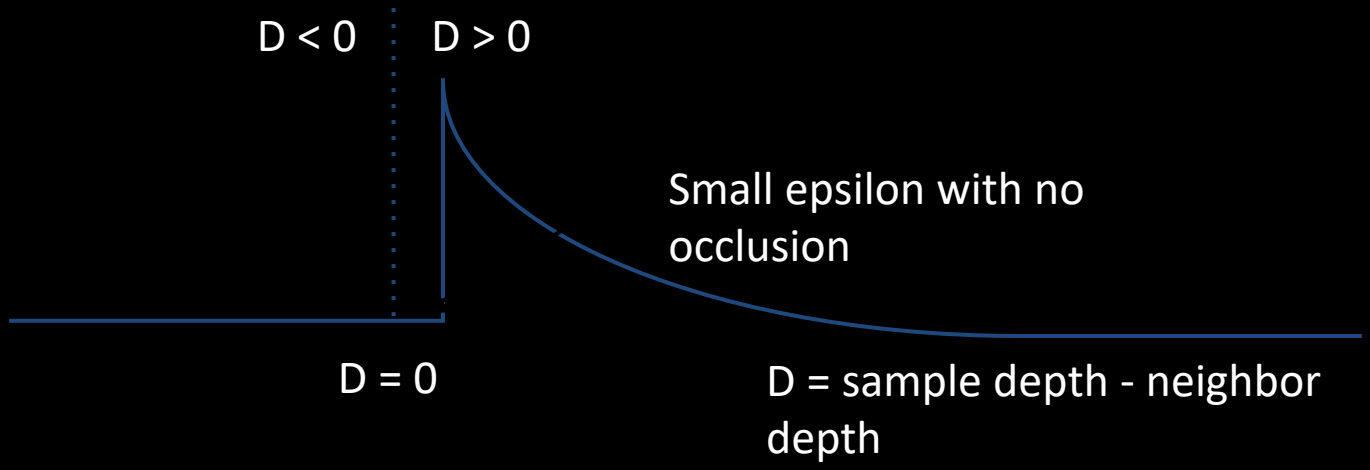


Screen-Space Ambient Occlusion

- Calculating occlusion
  - Probe nearby geometry using raycasting
  - Shoot rays in a hemisphere out of surface
  - Objects in close proximity cause darkening
- Idea: per-pixel occlusion approximation
  - Flatten raycasting to 2D in the image plane
  - Sample the depth of 8 to 32 neighboring pixels (requires deferred shading)
  - Don't count off-image samples as occlusions
  - Compare neighbor depth to 3D sample depth
  - If neighbor is in front, occlusion may be occurring

Screen-Space Ambient Occlusion

- Occlusion falloff function
  - Blockers closer to the sample should occlude more
  - Blockers far from the sample don't occlude at all
  - Blockers behind don't occlude at all



Real Time Local Reflections

- Used in Crysis 2

Real Time Local Reflections

- Used in Crysis 2

Real Time Local Reflections

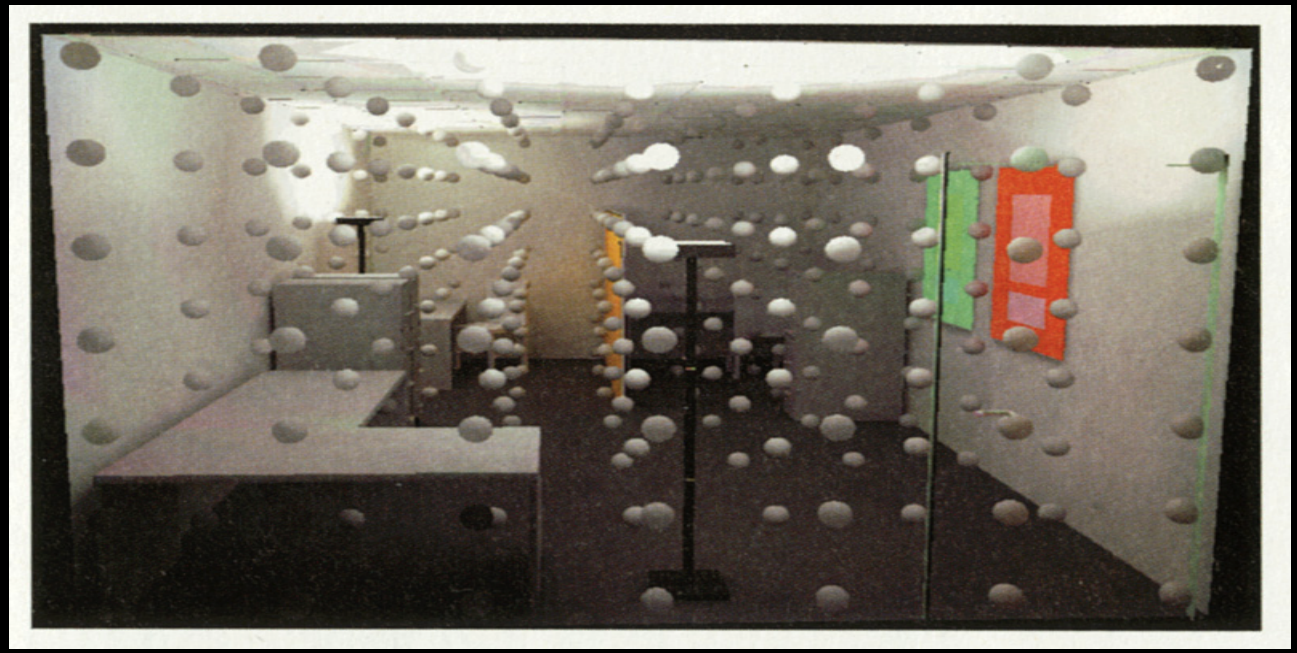
- Existing reflection methods (rasterization)
  - Render scene flipped about a plane
    - Use rendered scene as reflection
    - Only works for planar surfaces
  - Render scene from a point of view into cube map
    - Look up into cube map using reflection vector
    - Only works for small objects
- Reflections are expensive with rasterization
  - Need to render scene once per planar surface or per reflective object
  - Raytracing is much more straightforward

Real Time Local Reflections

- Raytrace reflections in screen space
  - Compute reflection vector per pixel using depth and normal from G-buffer
  - Raymarch along reflection vector
  - Project ray into 2D and check if scene depth is within threshold of ray depth
  - If so, use color from previous frame as reflection
- Edge cases (no data)
  - Fade out as reflection faces the camera
  - Fade out reflections off screen edge

Real Time Global Illumination

- Precomputed with irradiance cache

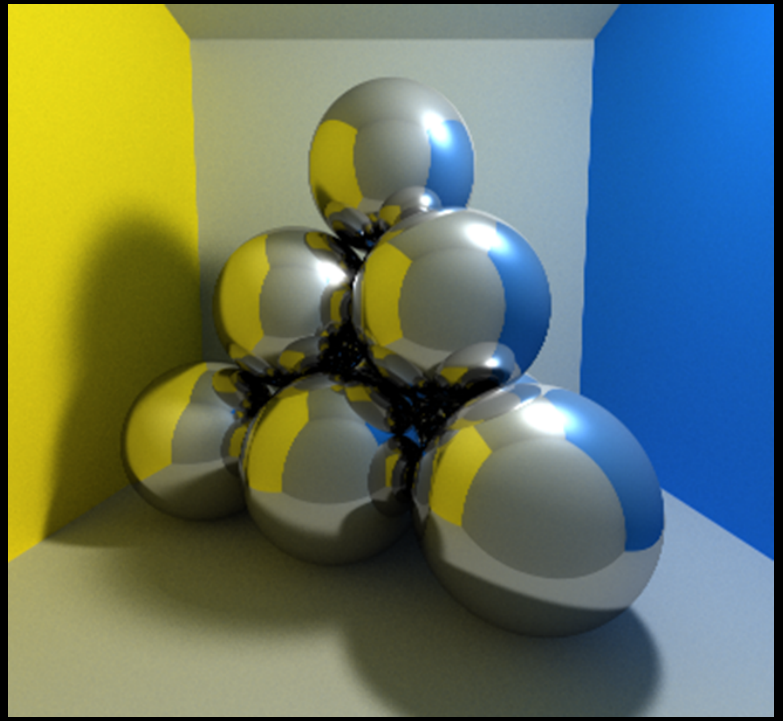


Real Time Global Illumination

- Precomputed with irradiance cache
  - Lightmaps for static objects
  - 3D grid of irradiance samples for dynamic objects
    - Each sample is snapshot of all light coming into a point
    - Think cube map, usually compressed using spherical harmonics
- Animated lightmaps
  - Static scene with moving light restricted to a path
- Precomputed Radiance Transfer (PRT)
  - Lightmap that can be queried by incident light angle
  - Lighting solution stored compressed using SH

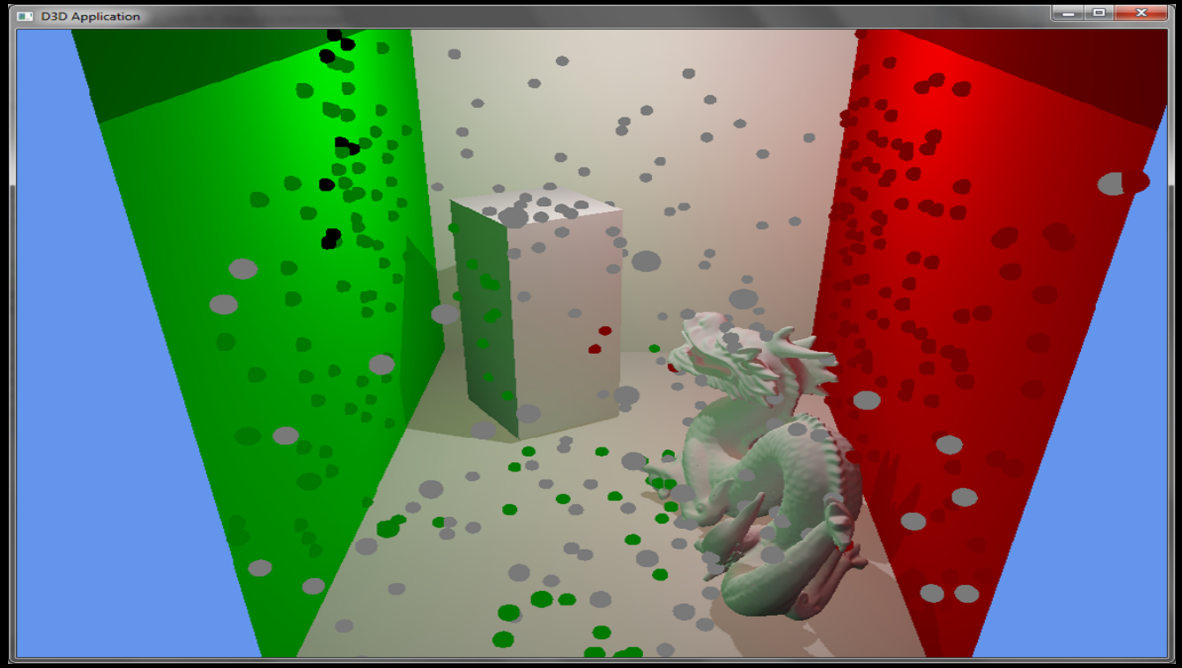
Real Time Global Illumination

- Path tracing directly
  - Not used much in games, technology still advancing



Real Time Global Illumination

- Instant radiosity



Real Time Global Illumination

- Instant radiosity
  - Shoot some photons into the scene (~200)
  - Only do one bounce
  - Virtual Point Light (VPL) where they land
- Rendering
  - Direct lighting: as usual (shadow maps)
  - Indirect lighting: each VPL becomes point light
- Updating
  - Randomized nature means lots of noise
  - Cache valid VPLs between frames


Class 8
C++ Tip of the Week

C++ Tip of the Week

LAMBDAS

Lambdas

- Inline functors:

```
int i = ...;
double terrainHeight = ...;
auto adjustEntityHeight = [terrainHeight, i](Vector3 &pos) {
    pos.y = terrainHeight*complicatedMath();
};
... // code that would use this math multiple times
```

```
// template
auto functionName = [capturedVariables](input arguments) { ... };
```

Lambda syntax

```
auto functionName = [capturedVariables](input arguments) { ... };
```

- auto tells the compiler to determine the type
  - in this case, it's a generated lambda definition
  - auto only works like this starting in C++11
    - Before that it declared a variable as 'local' storage, and now is almost never used
- Captured variables give the function access to variables in the surrounding scope

More lambda syntax

- To capture member variables, pass this as a captured variable

```
auto func = [this](int intput) { ... };
```

- You can optionally specify the return type for clarity

```
auto functionName = [capturedVariables](input arguments) ->
returnType { ... };
```

Even more lambda syntax

```
auto func = [&localOne, localTwo](int intput) { ... };
```

- localOne is passed by reference, whereas localTwo is passed as a copy (by default)

- Use '=' or '&' to capture all local variables in scope

```
auto func = [&](int intput) { ... }; // captures all variables by reference
auto func = [=, &localTwo](int intput) { ... }; // captures all variables
// by copy, except localTwo which is captured by reference
```

Using Lambda anonymously

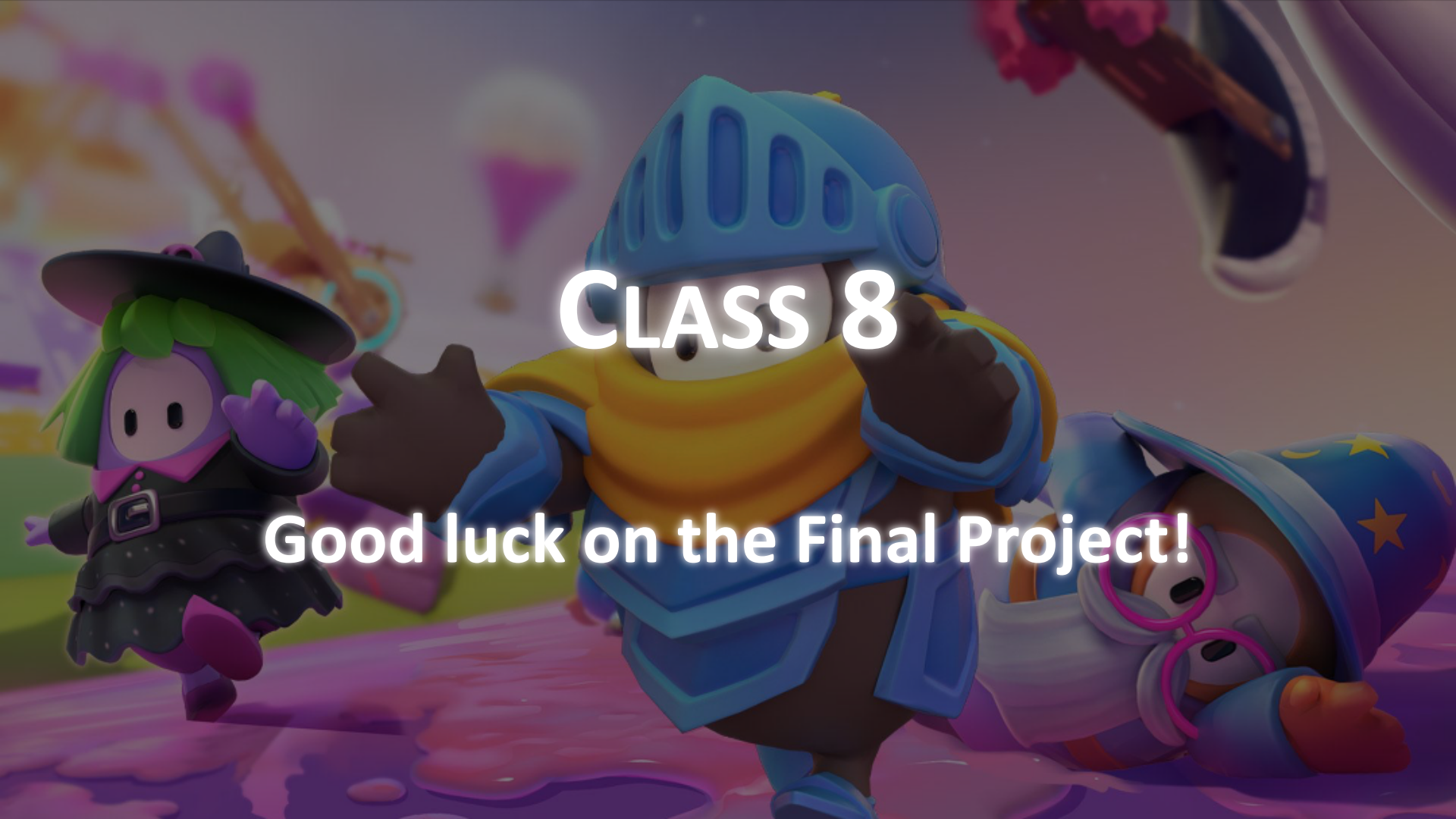
- As you probably guessed from the name lambda, you can also use them completely anonymously:

```
std::vector<int> some_list{ 1, 2, 3, 4, 5 };
int total = 0;
std::for_each(begin(some_list), end(some_list), [&total](int x)
{
    total += x;
});
```

Lambda based algorithms

- The standard library contains a bunch of helpful functions that make use of lamdas and lists
- Besides for_each, some examples are transform, count_if, remove_if, and binary_search

- Read more about lambdas: https://stackoverflow.com/questions/7627098/what-is-a-lambda-expression-in-c11


Class 8
Good luck on the Final Project!