Blending & Transparency

And the depth buffer

The Depth Buffer

Depth buffer

- Really just another texture – 2D grid of data
- Each “pixel” holds a single value
  - The “depth” of each pixel
  - Distance between camera’s near & far planes
- Used by rendering pipeline *automatically*
  - If set up properly
  - Starter code creates one for you
- Optional, but very useful

Depth buffer example

Depth buffer example

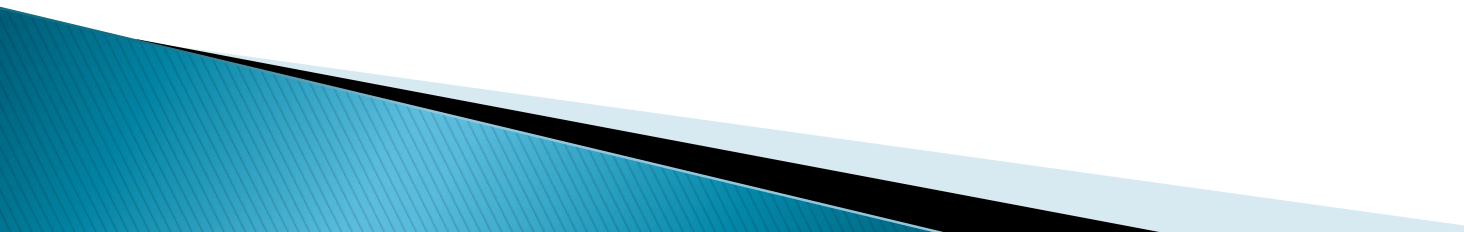
- Depth buffer has a single float value per pixel

What does a depth buffer do?

- Handles basic occlusion for us
  - “Occluded” pixels are discarded
  - Not actually drawn
- How is an “occluded” pixel defined?
  - Pixel with a depth >= an existing pixel
  - Only allows pixels to be drawn if they’re “closer”
- Can customize this behavior
  - By changing the *Depth/Stencil State*
  - Common to change for some advanced effects

Depth buffer – End result

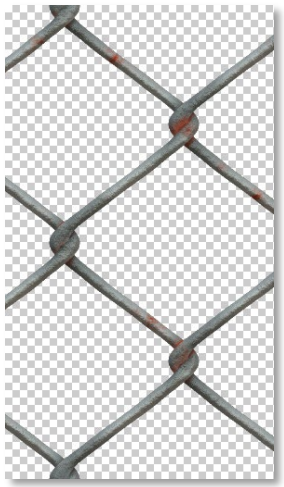
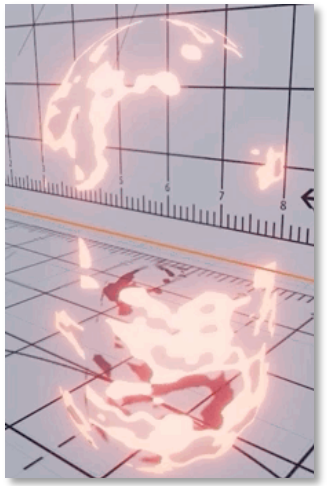
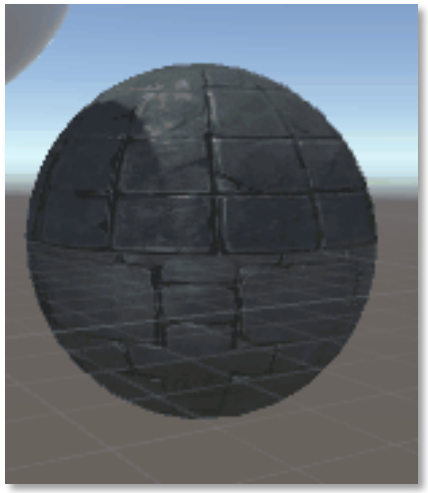
- We can draw 3D objects *in any order*
  - We don't have to sort them ourselves
  - Great when objects are *not transparent*
- Also handles intersecting geometry
  - Occlusion is handled *per-pixel*
  - Instead of per-triangle



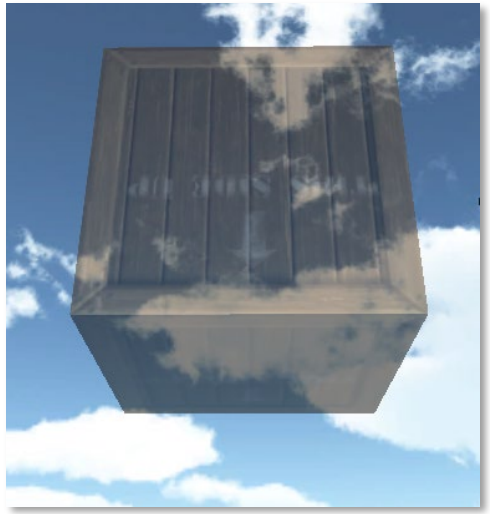
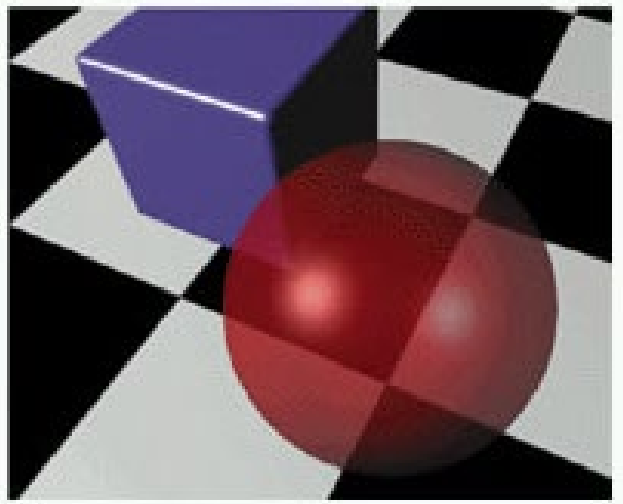
Blending & Transparency

Two categories of transparency

- Alpha clipping
  - 1-bit transparency
  - Each pixel is on or off
  - Per-pixel visibility
  - Sometimes called “Alpha Cutout”

- Blending
  - A combination of two colors
  - Each pixel has an alpha value that is used for blending

Blending examples

No blending

Alpha clipping

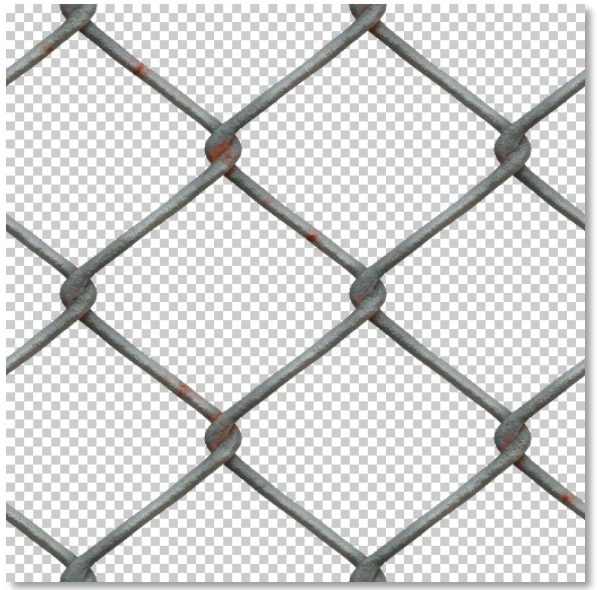
Alpha blending

Alpha Clipping

AKA “Alpha Cutout”

Alpha clipping

- Simple on/off transparency
  - 1-bit transparency
- Like chain-link fence or leaves
- No actual blending necessary
  - 100% or 0%
  - Keep pixel or don't

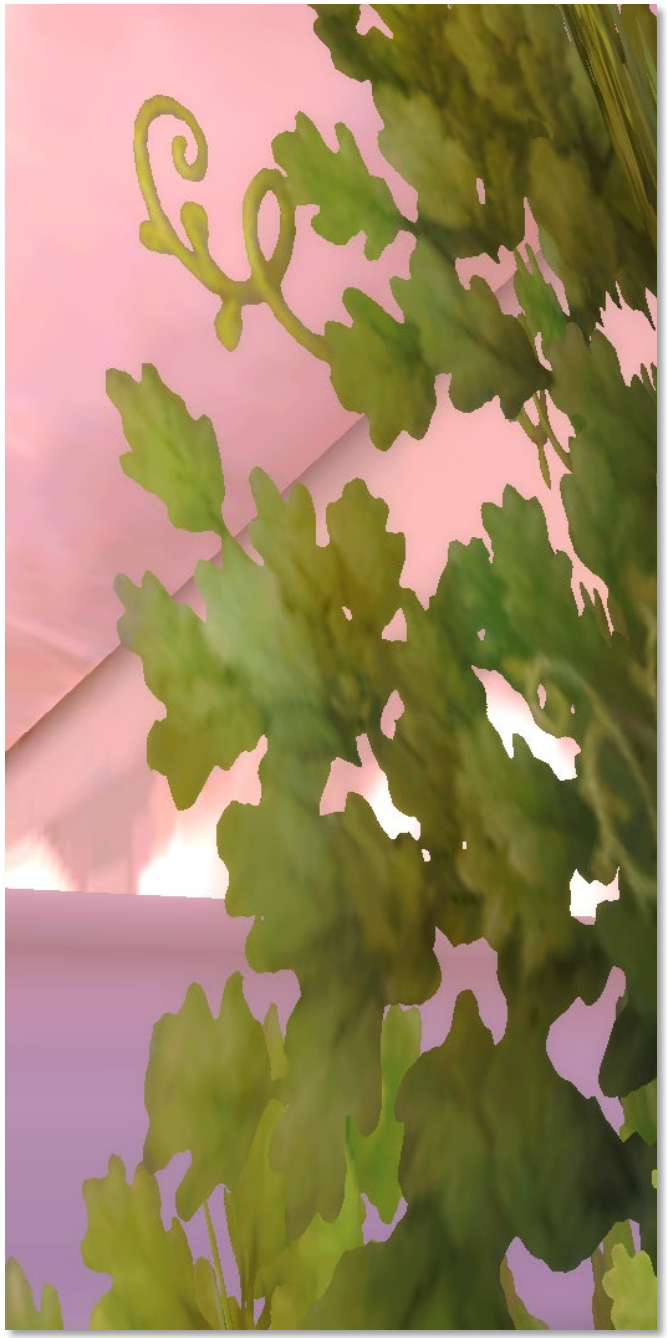


Discarding pixels

- Pixel shader can completely discard a pixel
  - Shader immediately ends
  - No color output
  - Even better: *no depth buffer output!*

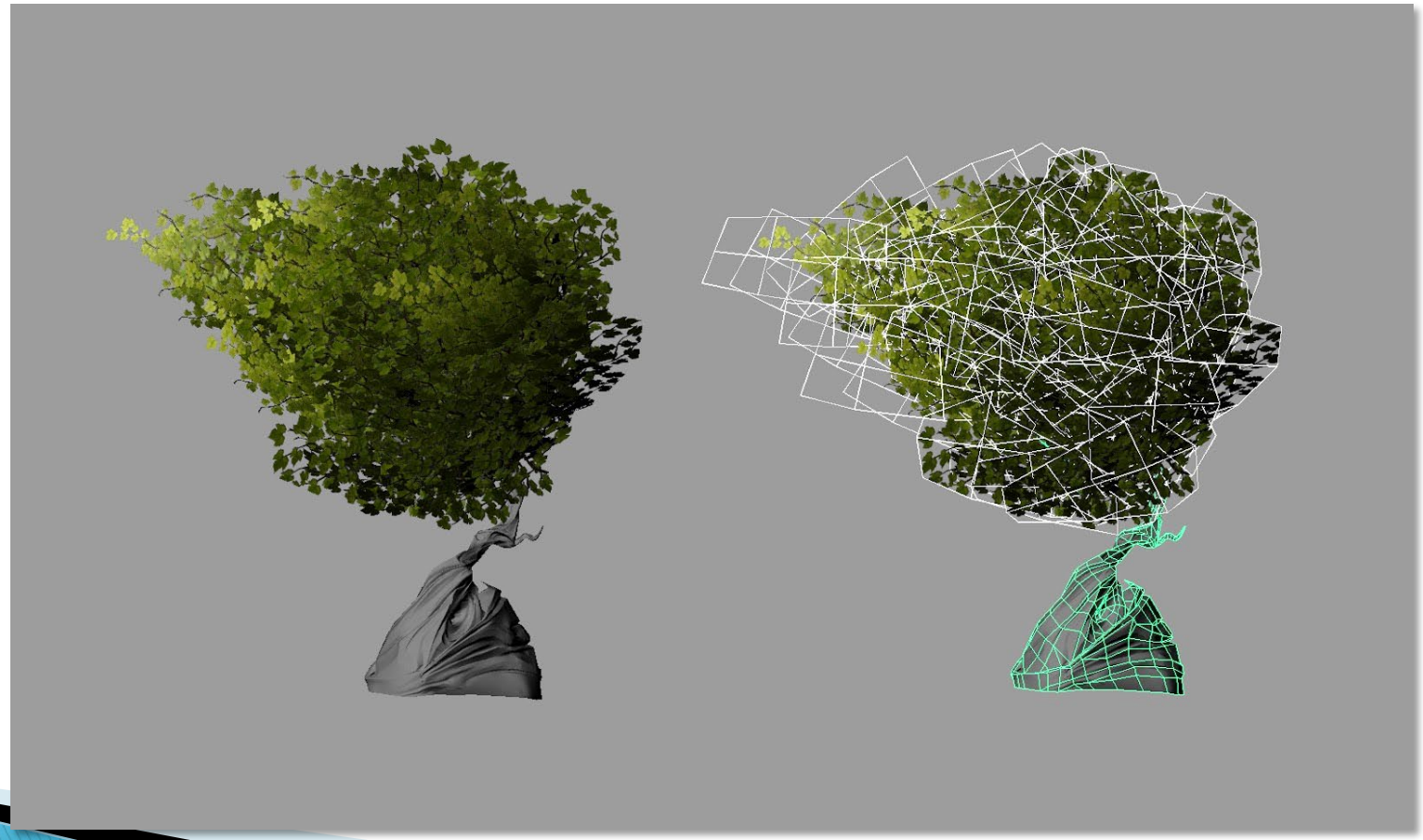
Clipping close up

- Texture applied to very few polygons
- “Transparency” controlled by texture’s alpha
- Leaves are **not** modeled individually!

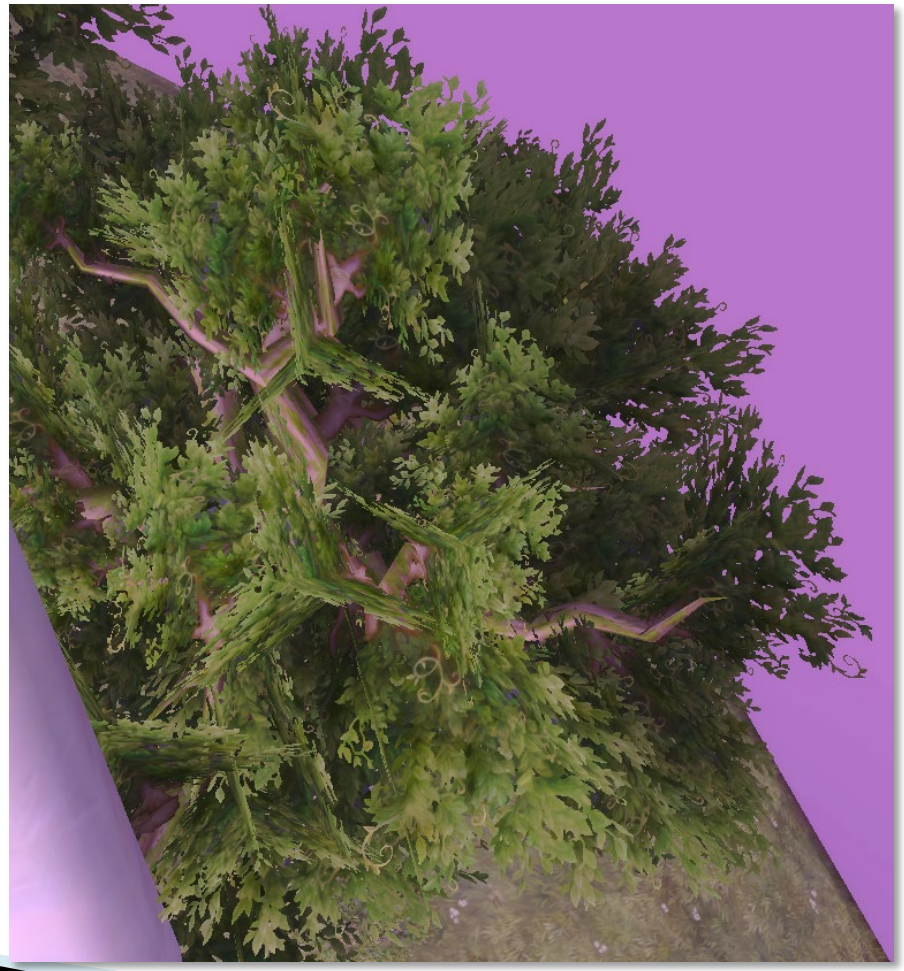


Alpha clipping for trees/foliage

- Thick foliage with minimal polygons



Foliage in WoW: Top-down



Discarding pixels in HLSL

- “`discard;`” statement

- Immediately ends the pixel shader
  - No color output
  - And *no depth buffer output!*

```
if( color.a < 0.1f )
    discard; // Discard below 0.1 alpha
```

Conditional discard with clip()

```
clip(someValue); // If someValue < 0, pixel is discarded
```

- Example: Clip anything below 10% alpha

```
// Discards below 0.1 alpha
clip(color.a - 0.1f);
```

Implementing alpha clip

- Simply create (and use) a new shader
  - No alpha clipping? Use existing shader
  - Alpha clipping? Use new shader
- Control which is applied via material system
- No other C++ changes necessary
  - Although you could pass in the “clip” level/amount
  - As an external shader variable

Blending

Blending – Combining colors

- Most often used for transparency

Object being rendered (50% alpha)

Existing colors (previously rendered)

Desired results

Blending is OFF by default

- Default is to simply replace existing colors
  - Replacing is faster than blending

Object being rendered (50% alpha)

Existing colors (previously rendered)

Actual results

How does blending work?

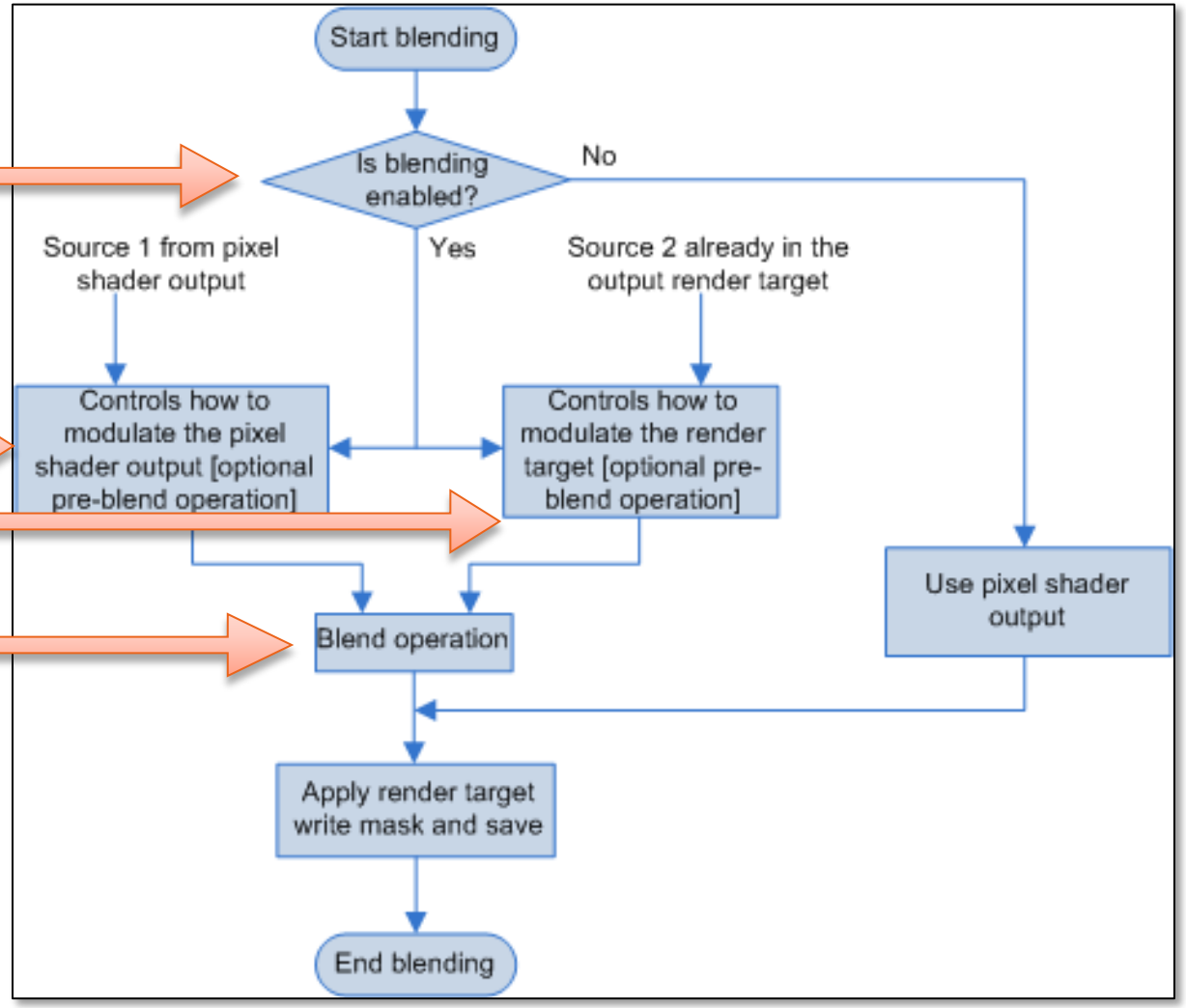
- Handled by GPU
- Occurs during Output Merger pipeline stage
  - If blending is enabled
  - And configured properly
- GPU combines two colors:
  - **Source Color**: Current pixel shader output color
  - **Destination Color**: Existing color in the render target
  - `finalColor = sourceColor * someValue + destinationColor * anotherValue`

Output merger's "Blend Stage"

Can enable or disable blending entirely

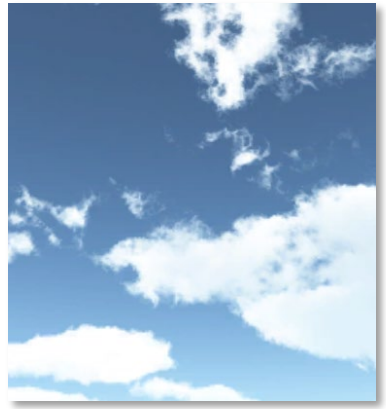
Can customize interpretation of **both** colors

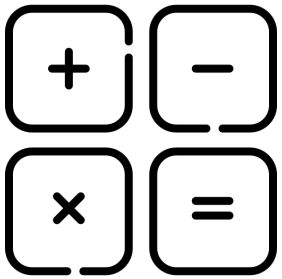
"Blend Operation" also customizable



Blend options

- Source Blend
  - How to interpret the source pixel color
  - Source is the NEW color – pixel being rendered
- Destination Blend
  - How to interpret the destination pixel color
  - The color that exists already in the render target
- Blend Operation
  - How to combine these values
  - Add, subtract, min, max, etc.

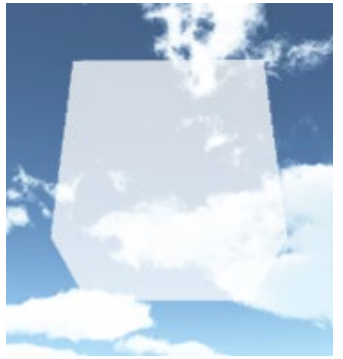




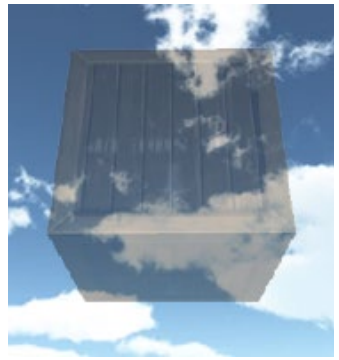
Blend states for alpha blending

- Source Blend:
  - D3D11_BLEND_**SRC_ALPHA**
  - Source modulated by it's own alpha value
- Destination Blend
  - D3D11_BLEND_**INV_SRC_ALPHA**
  - Dest modulated by (1.0 – SOURCE's alpha)
- Blend Operation = Add
  - D3D11_BLEND_OP_**ADD**

- alpha = 0.25
- Use 25% of these colors



- Use 75% of these colors



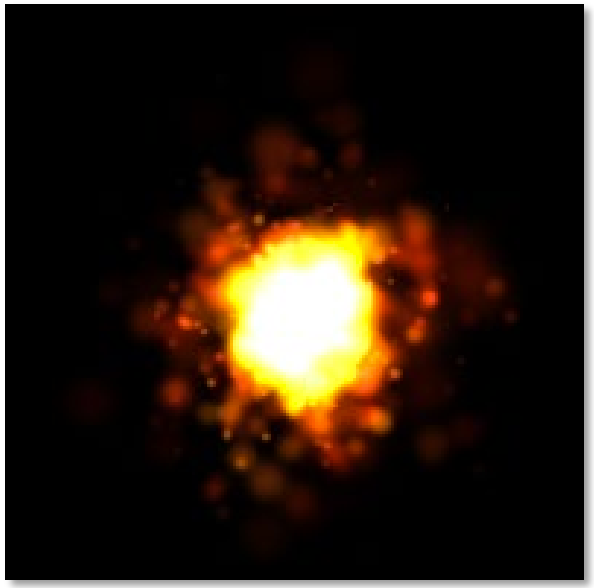
- Add results

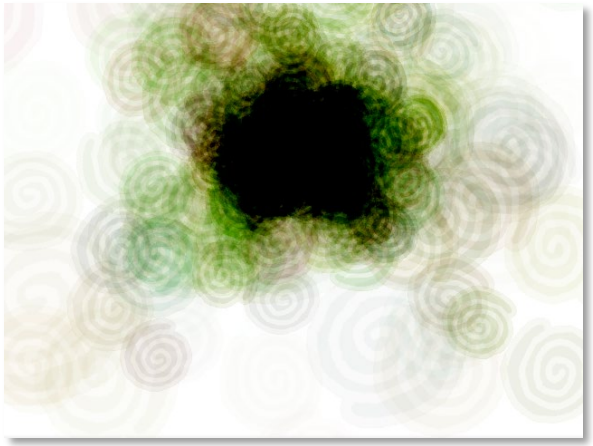
Other blending options?

- Mostly used for special effects

- Additive – Like fire
  - Adds colors together
  - Colors end up being brighter

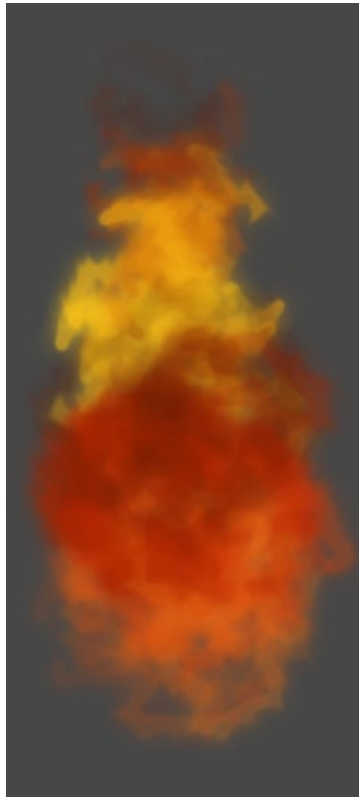
- Subtractive
  - Subtracts one color from another
  - Colors end up being darker
  - Not used as frequently as alpha/additive





Additive vs. alpha blending

- Additive blend
  - Full color of each pixel added together
- Only gets brighter



- Alpha blend
  - Colors blended by alpha
- Colors overlap

(Note: Depth writing is usually **turned off** for particles)

Blend states for additive blending

- Source Blend = ONE
- Dest Blend = ONE
- Blend Op = ADD

- What's happening?
  - Full colors are simply added together
  - Result = SrcColor * 1 + DestColor * 1

- Uses
  - Particle systems – Fire, lasers, sparkles
  - Anything that should get brighter/glow with overlap

Enabling Blending

In Direct3D

Enabling blending

- Blending options defined by a Blend State
  - `ID3D11BlendState*`
- Standard Direct3D set-up:
  - Define an `ID3D11BlendState` pointer
  - Fill out `D3D11_BLEND_DESC`
  - Call `device->CreateBlendState()`
- Change state as necessary
  - `context->OMSetBlendState()`

D3D11_BLEND_DESC members

- RenderTarget[]
  - Array of 8 D3D11_RENDER_TARGET_BLEND_DESC's
  - Allows custom blending for up to 8 targets
  - Fill out the first one of these (see next slide)

- IndependentBlendEnable
  - Bool – Will each target have it's own settings?
  - If false, first element of above array used for all targets

- AlphaToCoverage
  - Bool – Used by advanced multi-sampling techniques

D3D11_RENDER_TARGET_BLEND_DESC

- BlendEnable

- SrcBlend
- DestBlend
- BlendOp

  - Handles blending RGB components
  - Necessary for transparency

- SrcBlendAlpha
- DestBlendAlpha
- BlendOpAlpha

  - Can blend alpha component separately
  - Not necessary for basic transparency

- RenderTargetWriteMask

D3D11_BLEND_DESC example

```
D3D11_BLEND_DESC bd = {};

bd.AlphaToCoverageEnable          = false;
bd.IndependentBlendEnable         = false;
bd.RenderTarget[0].BlendEnable    = true;
bd.RenderTarget[0].SrcBlend       = D3D11_BLEND_SRC_ALPHA;
bd.RenderTarget[0].DestBlend      = D3D11_BLEND_INV_SRC_ALPHA;
bd.RenderTarget[0].BlendOp        = D3D11_BLEND_OP_ADD;
bd.RenderTarget[0].SrcBlendAlpha  = D3D11_BLEND_ONE;
bd.RenderTarget[0].DestBlendAlpha = D3D11_BLEND_ZERO;
bd.RenderTarget[0].BlendOpAlpha   = D3D11_BLEND_OP_ADD;
bd.RenderTarget[0].RenderTargetWriteMask = D3D11_COLOR_WRITE_ENABLE_ALL;

ID3D11BlendState* blendState;
device->CreateBlendState(&bd, &blendState);
```

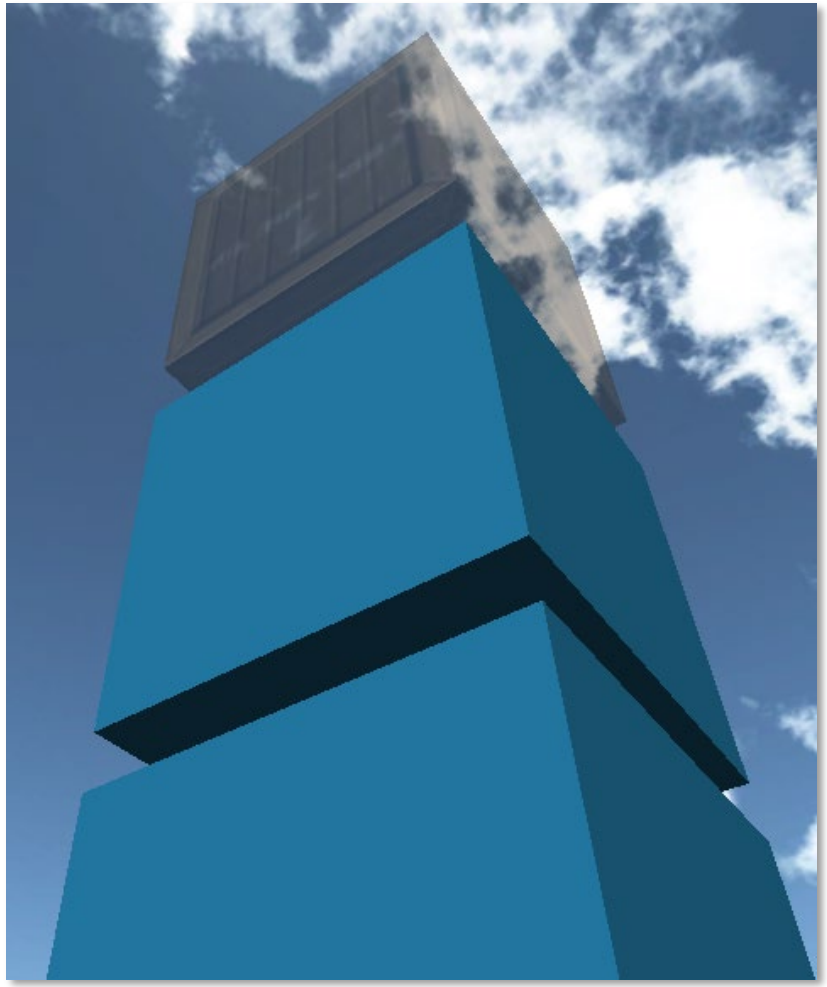
Blending Issues & Best Practices

Issue: Blending & depth

- Depth Buffer tracks pixel depths
  - Only “closer” pixels actually render
  - Can render (opaque) objects in any order
- Example
  - Cubes can be drawn in any order
  - Results will always be the same
  - If they’re opaque
  - Technically faster to render front-to-back however

Seeing through pixels

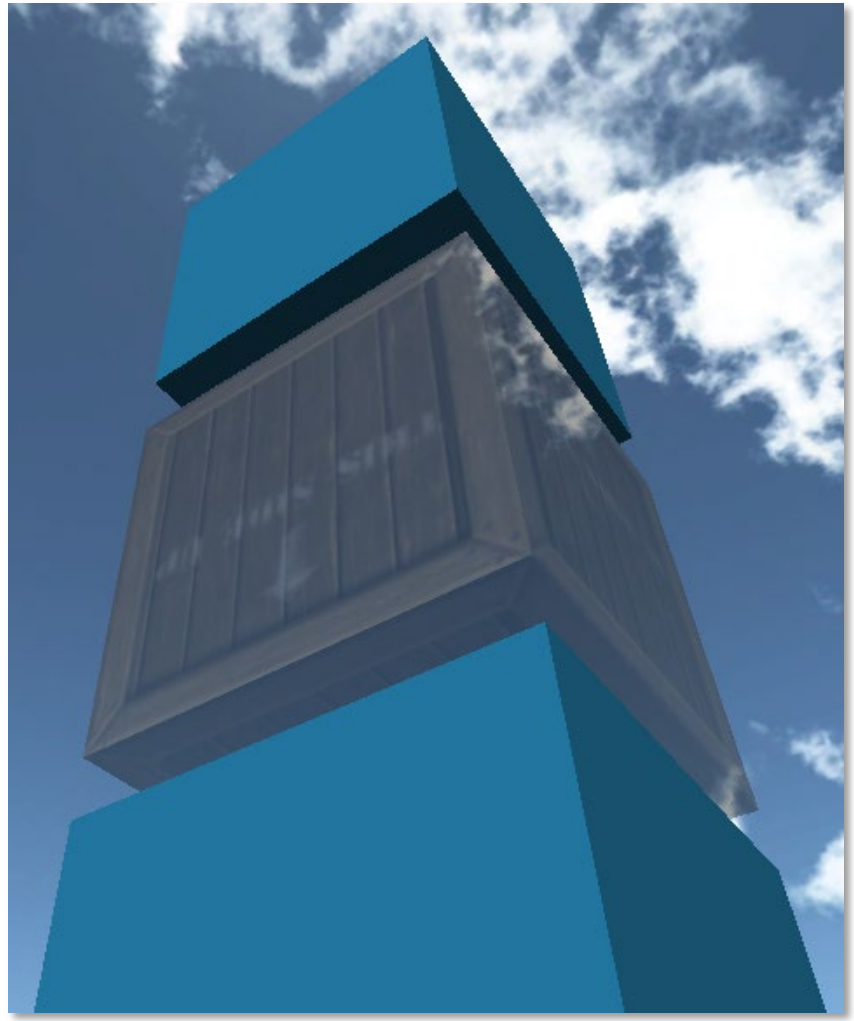
- Not like layers in photoshop!
- Blends w/ previously drawn pixels
  - But only if pixels *actually render*
- Example →
  - Sky drawn first
  - Drawing transparent cube next
  - Then blue cubes



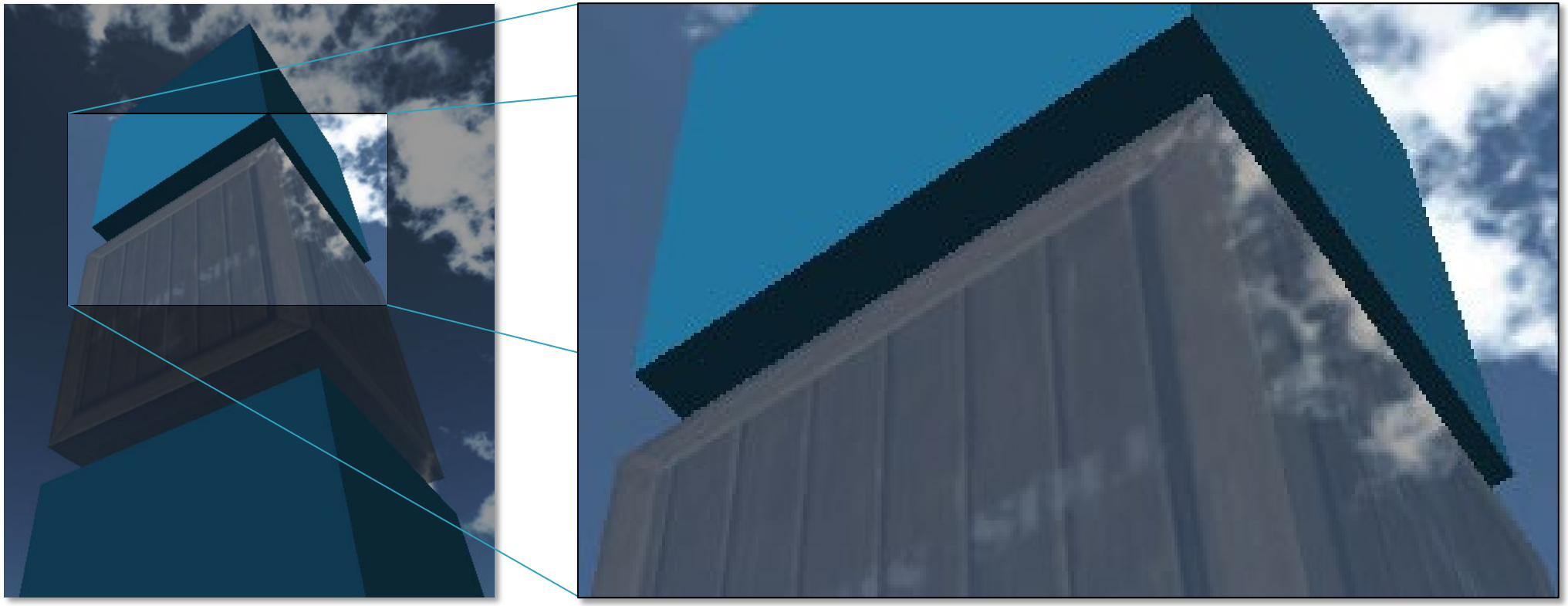
Draw order problem!

- Transparent pixels still have depth!
  - They still write to depth buffer
  - Can occlude objects drawn later

- Problematic Example →
  - Crate drawn BEFORE top cube
  - Top cube should be “behind” crate
  - Top cube pixels rejected due to depths

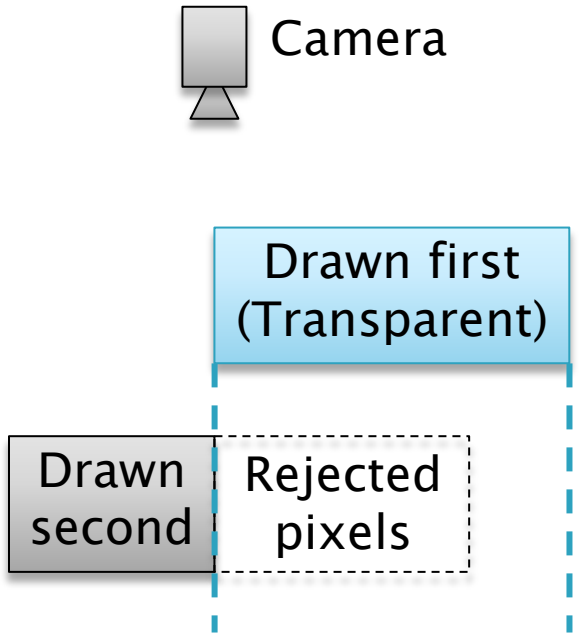


Draw order problem – Close Up



Blending & draw order

- *Closer* transparent object drawn first
- *Farther* object drawn second
- Transparent (closer) pixels block the second object's pixels



- Take away: Draw order matters with transparent objects!

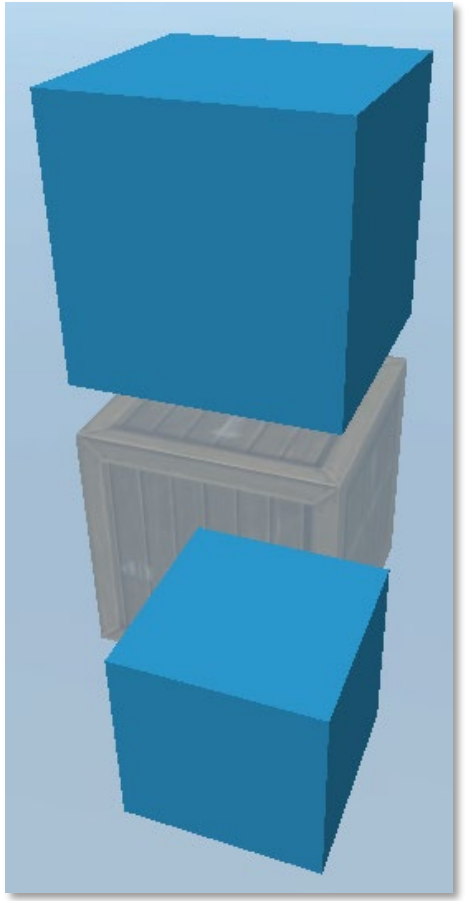
Draw order solution: Sorting!

- Sort your transparent objects
  - By distance to the camera
  - Every frame

- Correct steps for transparency:
  1. Sort transparent objects back-to-front
  2. Turn blending off → Draw all opaque objects
  3. Draw skybox
  4. Turn blending on → Draw sorted transparent objects

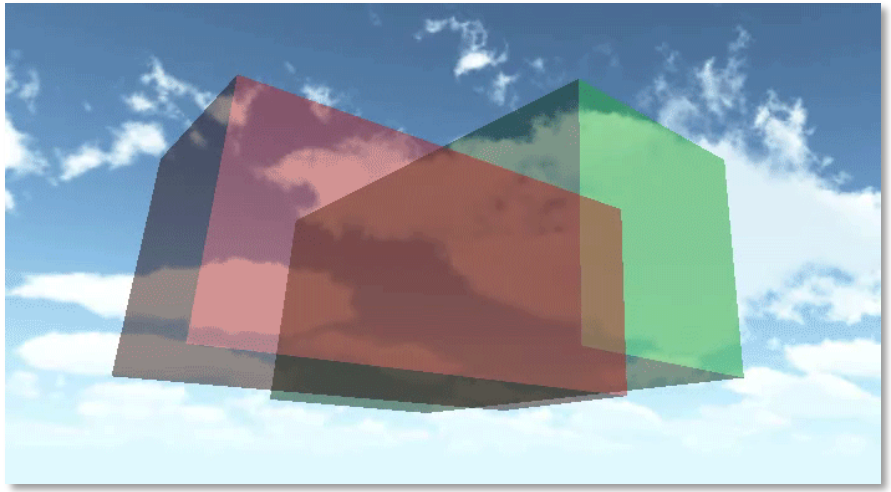
Draw order solution – No depth?

- Can we just disable the depth buffer instead of sorting?
- Nope!
- Example →
  - Without the depth buffer, newly drawn objects are always “on top” regardless of depth



Intersecting transparency

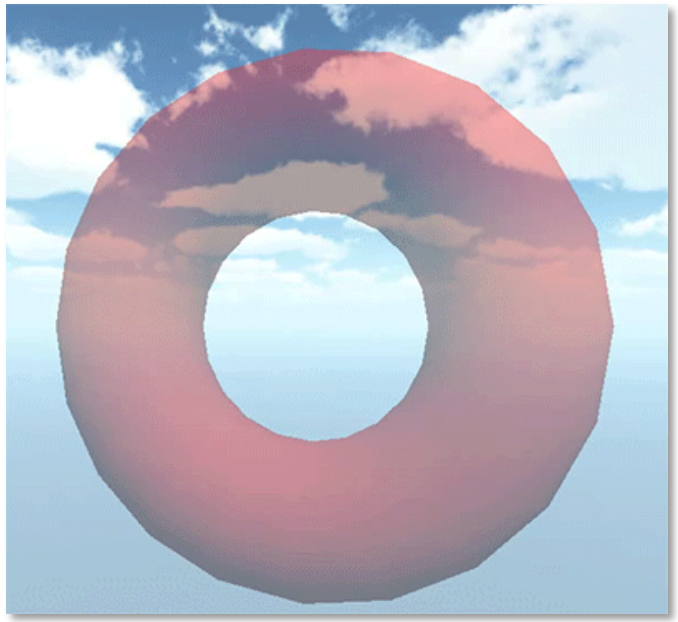
- Intersecting transparent objects problematic:
  - One whole object is drawn first



- Design game/models around this issue

Single mesh transparency

- Complex transparent meshes are problematic
  - Overlapping, front-facing triangles
  - Design game/models around this issue



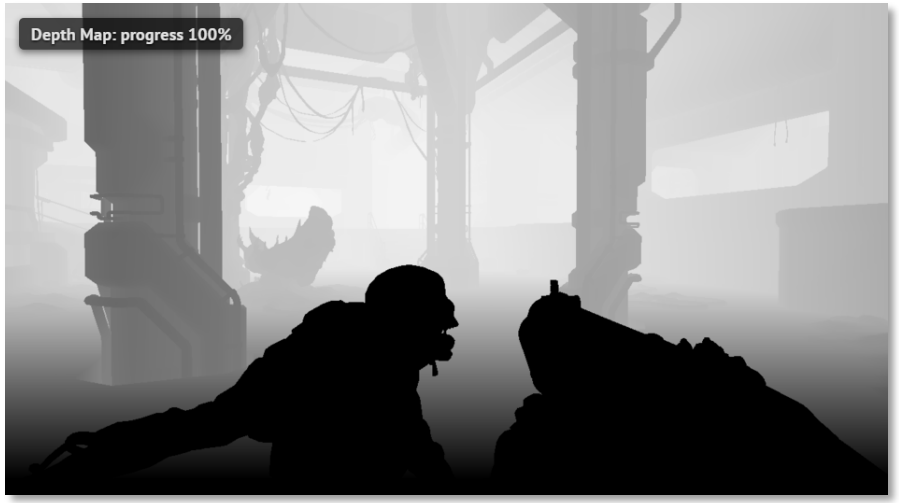
Best practices

- Transparency – Set and forget? NO!
  - Blending is slower than just overwriting pixels
  - Blending fully opaque objects is wasteful
- Changing render states has a cost (time)
  - Don’t change blend states between every draw
- Solution?
  - Turn blending **OFF,** then draw all solid objects (and sky)
  - Turn blending **ON,** then draw all transparent objects

Advanced Blending Techniques

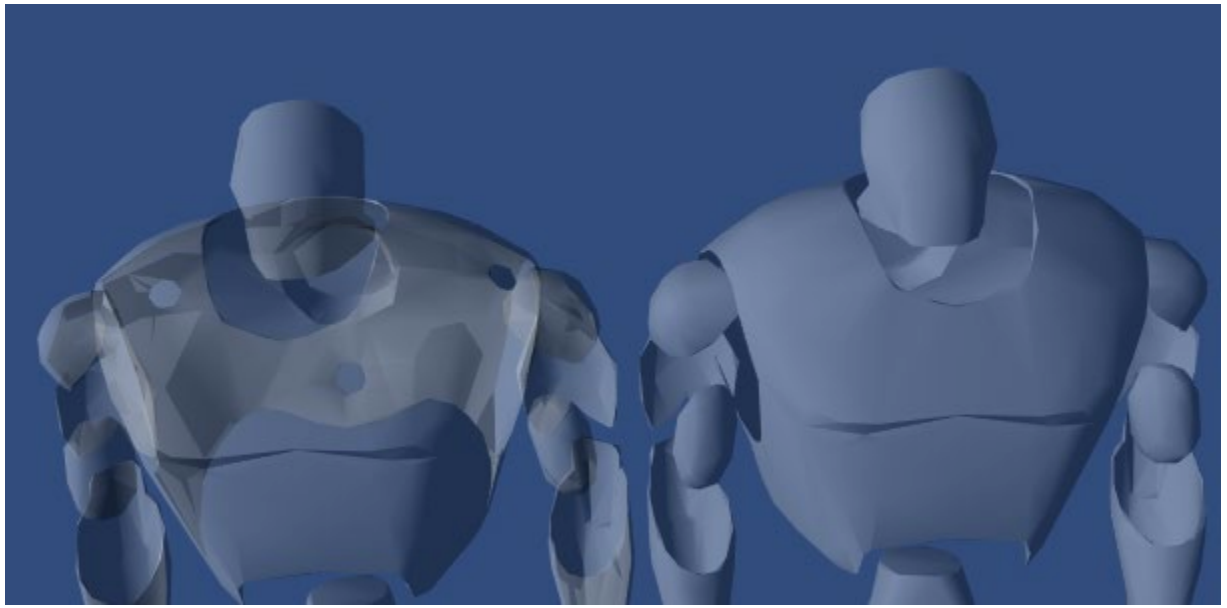
Depth Pre-Pass

- Rendering opaque surfaces to depth buffer
  - No pixel shader used at all
  - But requires 2x draw calls
- Ensures only closest opaque pixel is shaded
  - Great when pixel shader is expensive
  - Or when lots of overdraw can occur
- Many engines do this
  - Doom (2016)
  - Unity



Depth pre-pass for transparency

- Complex transparent objects have overlap
- Can pre-fill depth info w/ depth pre-pass



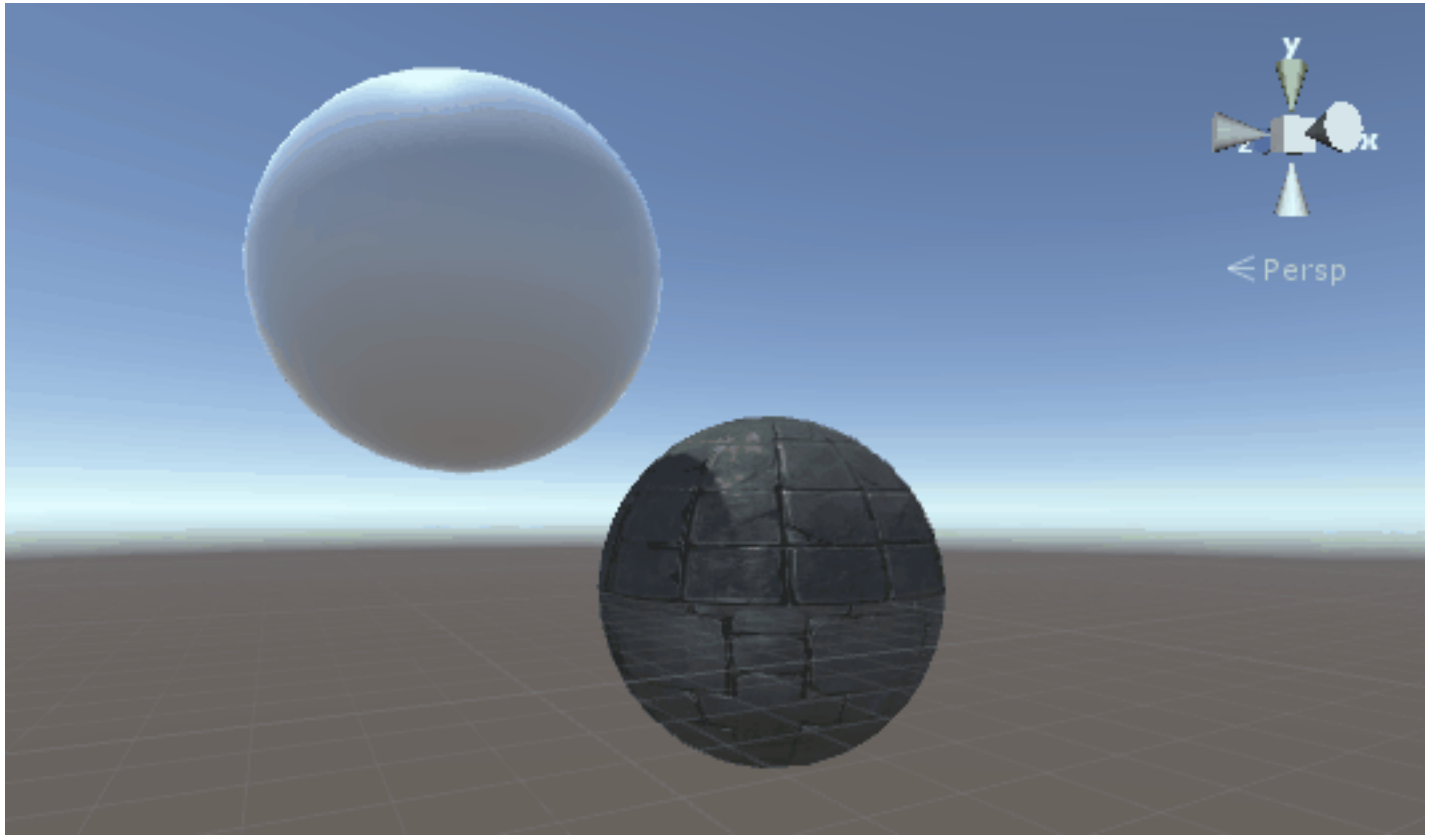
Blending pre-pass: WoW

- When camera gets close to your character
  - Notice you can see ground but not other shoulder

Avoiding blending entirely

- Blending is often used to fade objects in/out
  - Spawning new cars
  - Seeing through walls when character enters room
- Many new games avoid it entirely
- Instead “dither” an object in/out
  - Removing individual pixels over time
  - Until none are left

Dither Fade – Example



AKA: Screen-Door Transparency

XCOM 2 – Dither fade