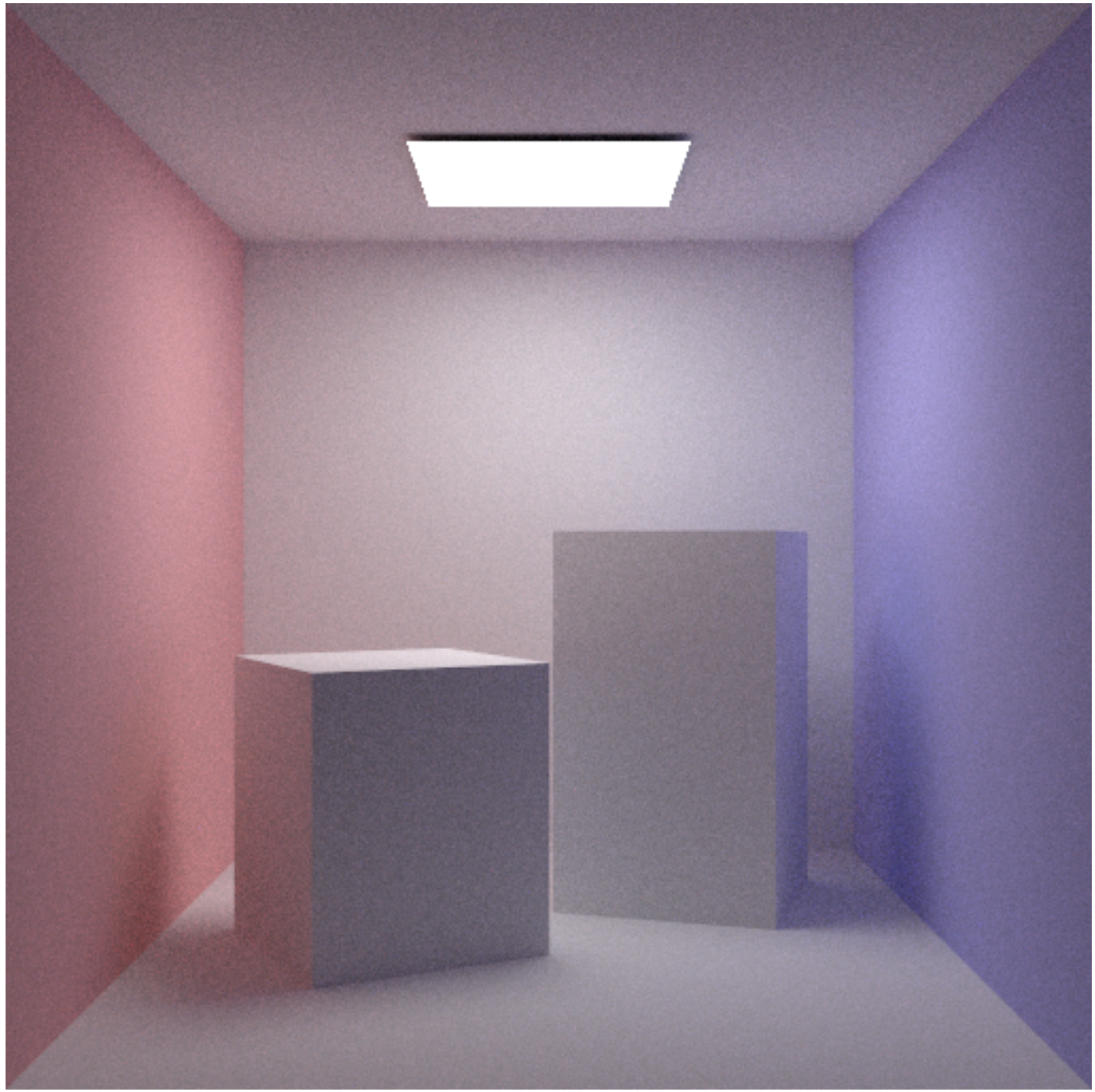
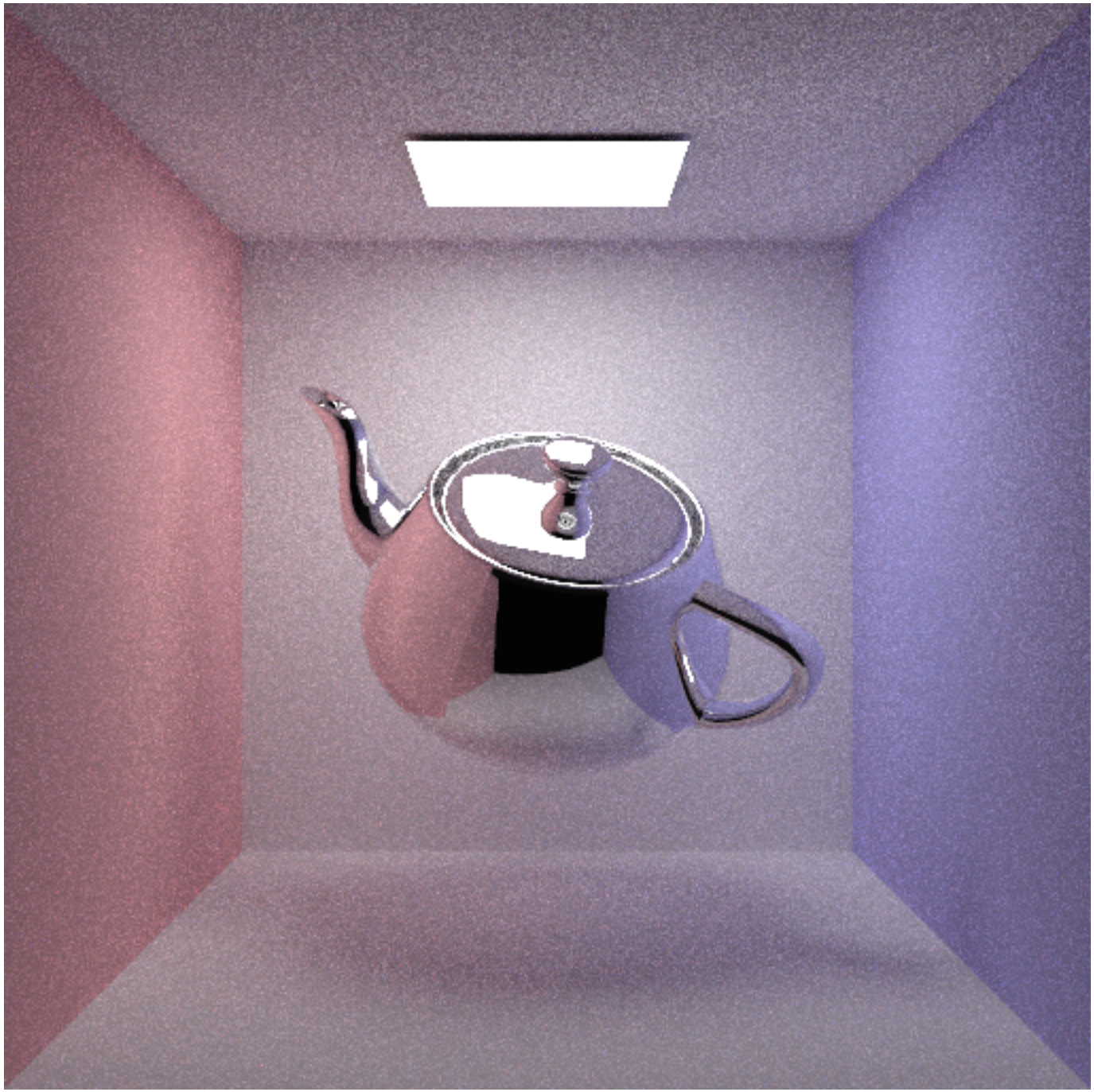
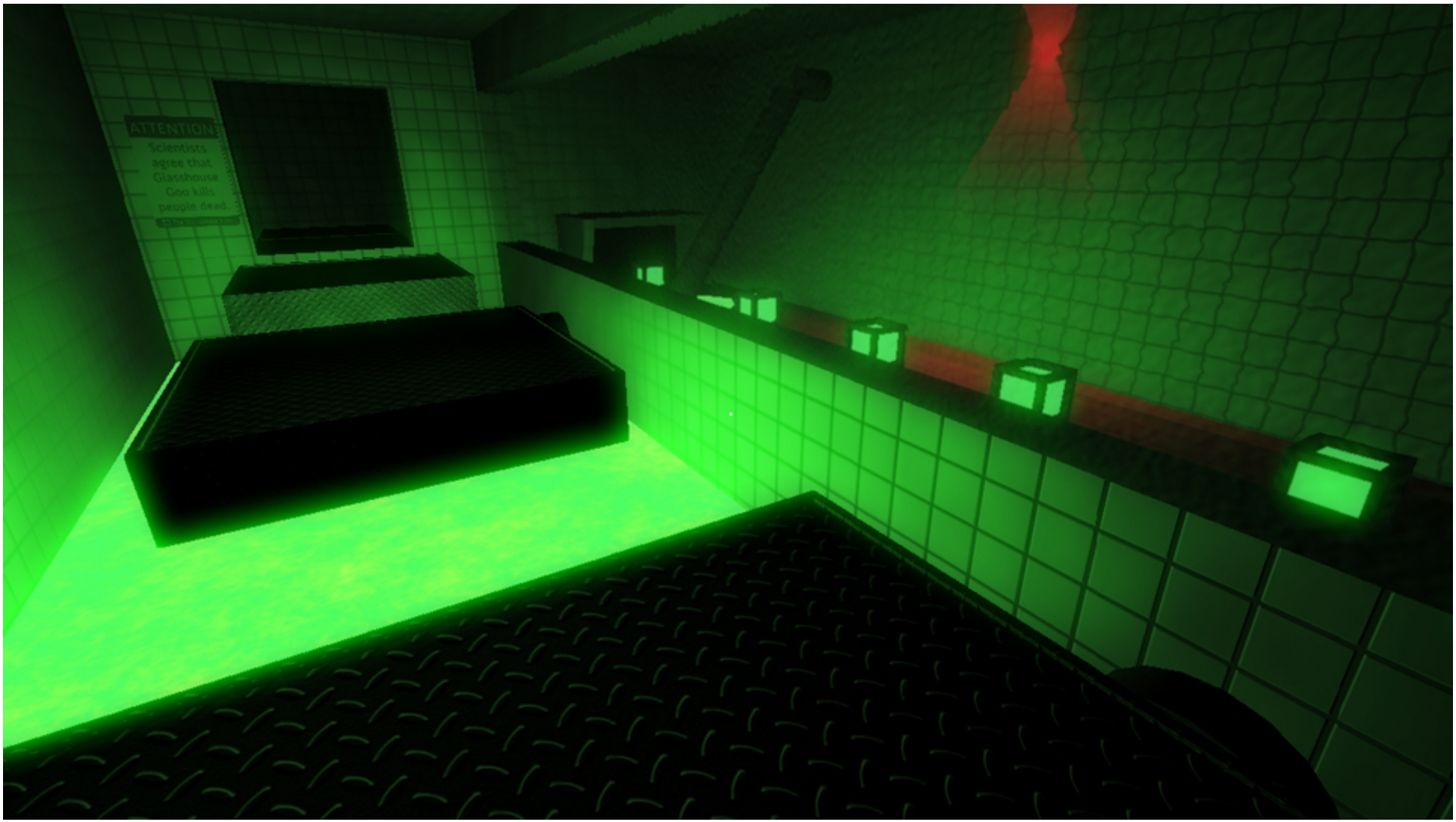
3D Graphics 101

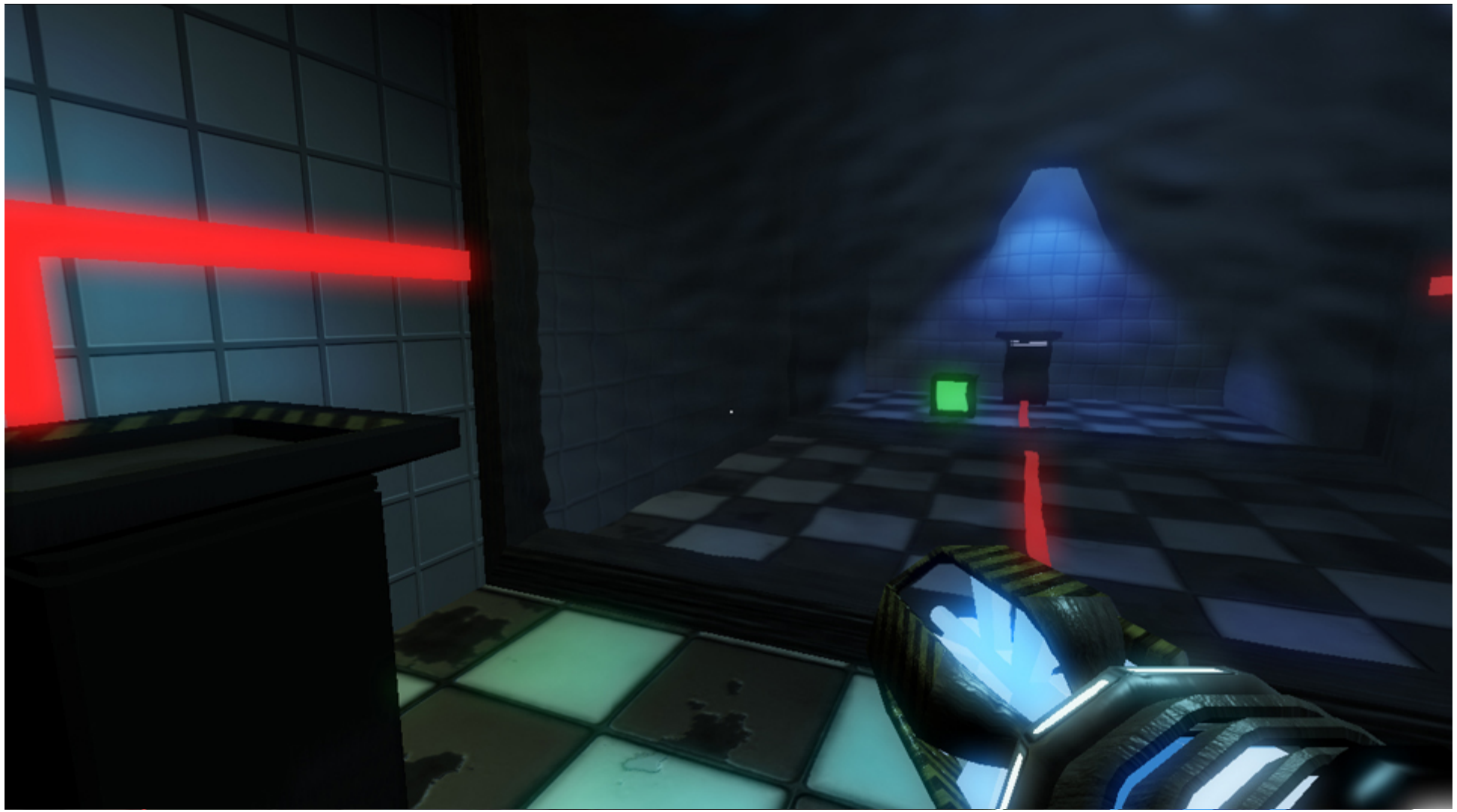
An introduction to the Graphics Rendering Pipeline

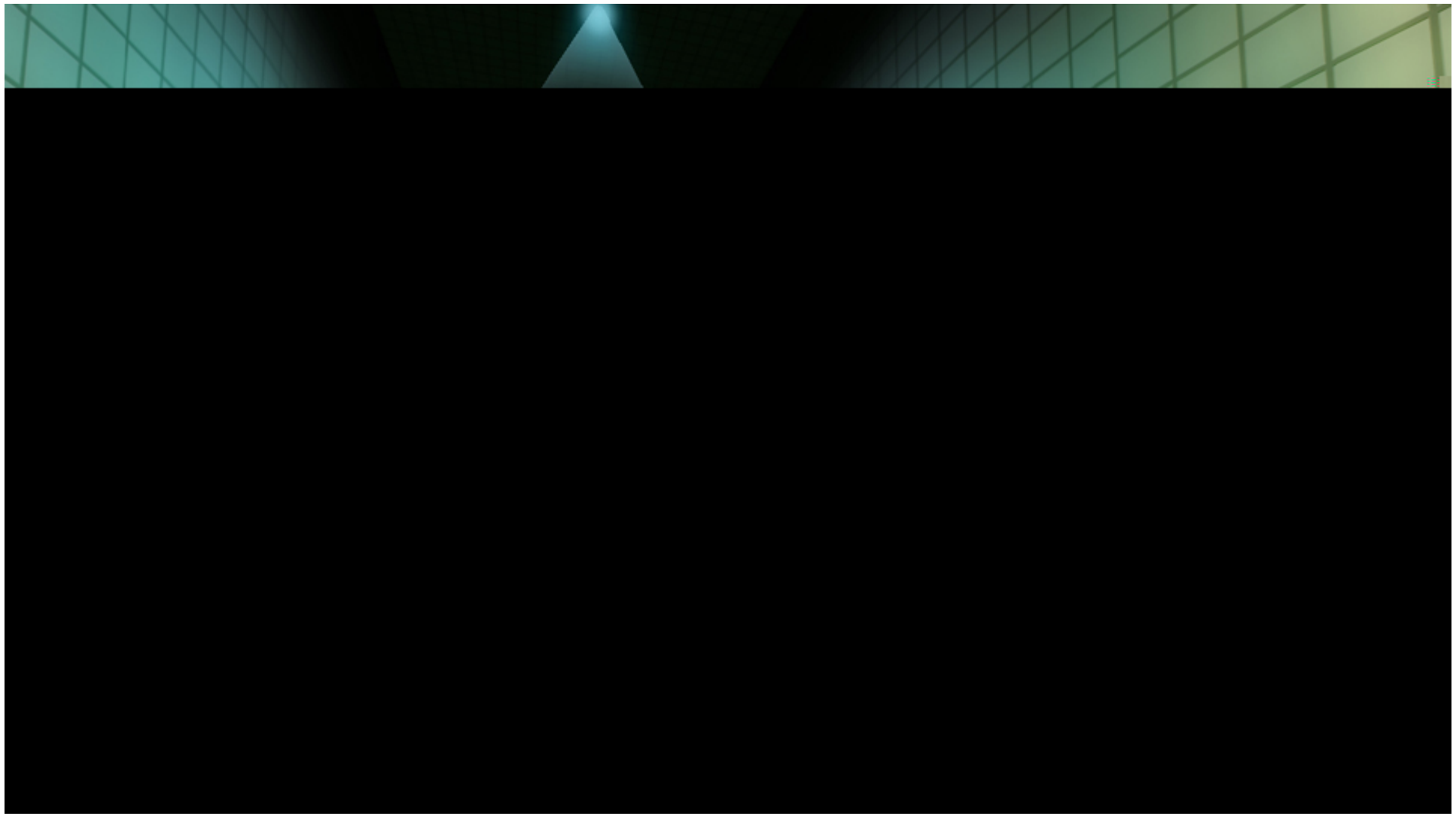


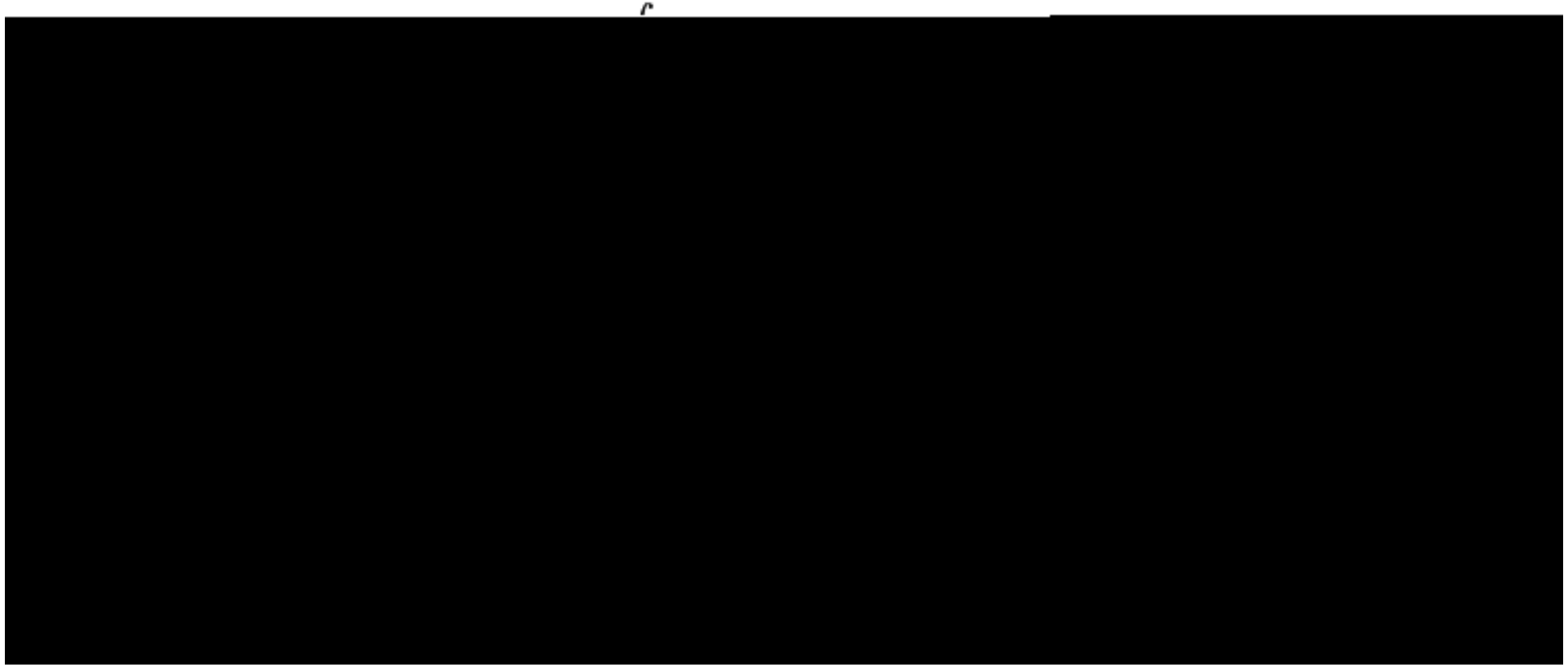




http://rocktopusgames.com/

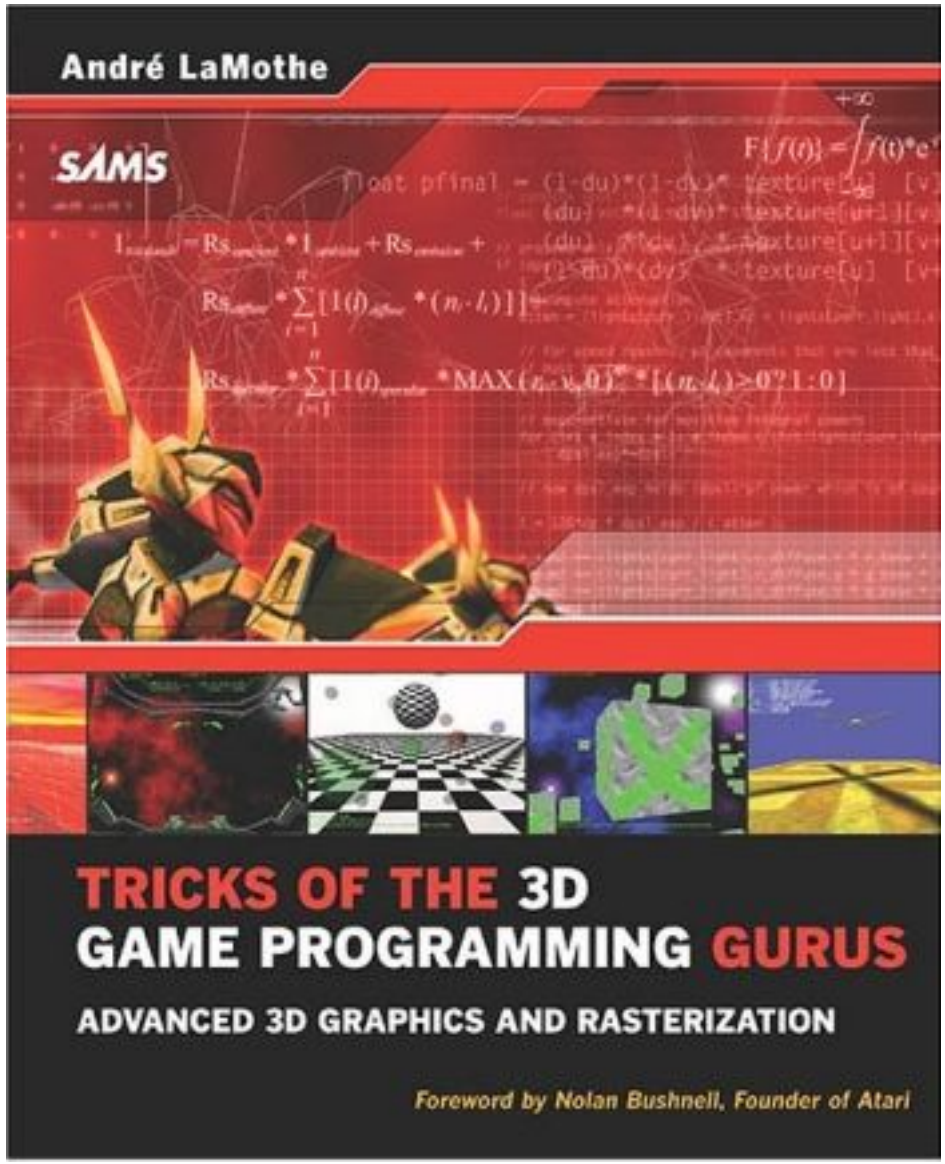




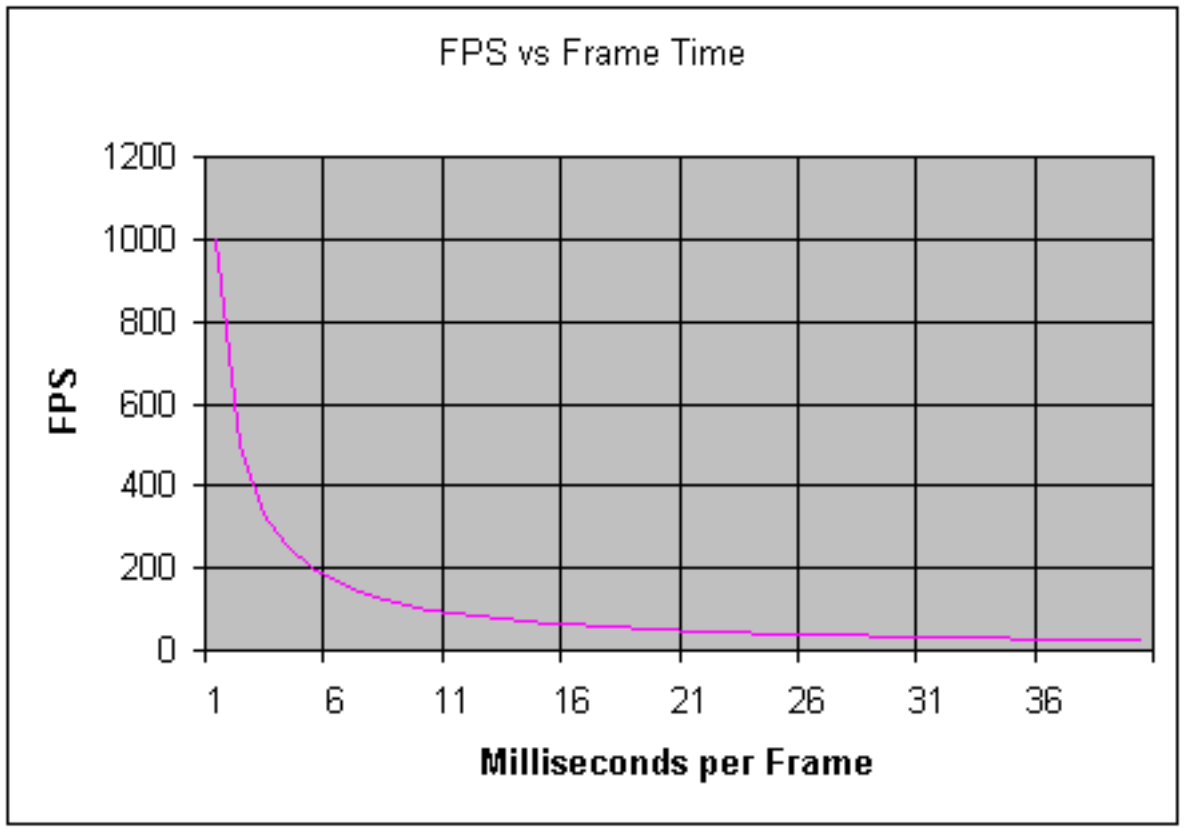


just kidding...

Many of the diagrams in this presentation come from the excellent book:



Frames Per Second is Non-Linear



1000ms/sec / 900 FPS = 1.111.. ms per frame

1000ms/sec / 450 FPS = 2.222.. ms per frame

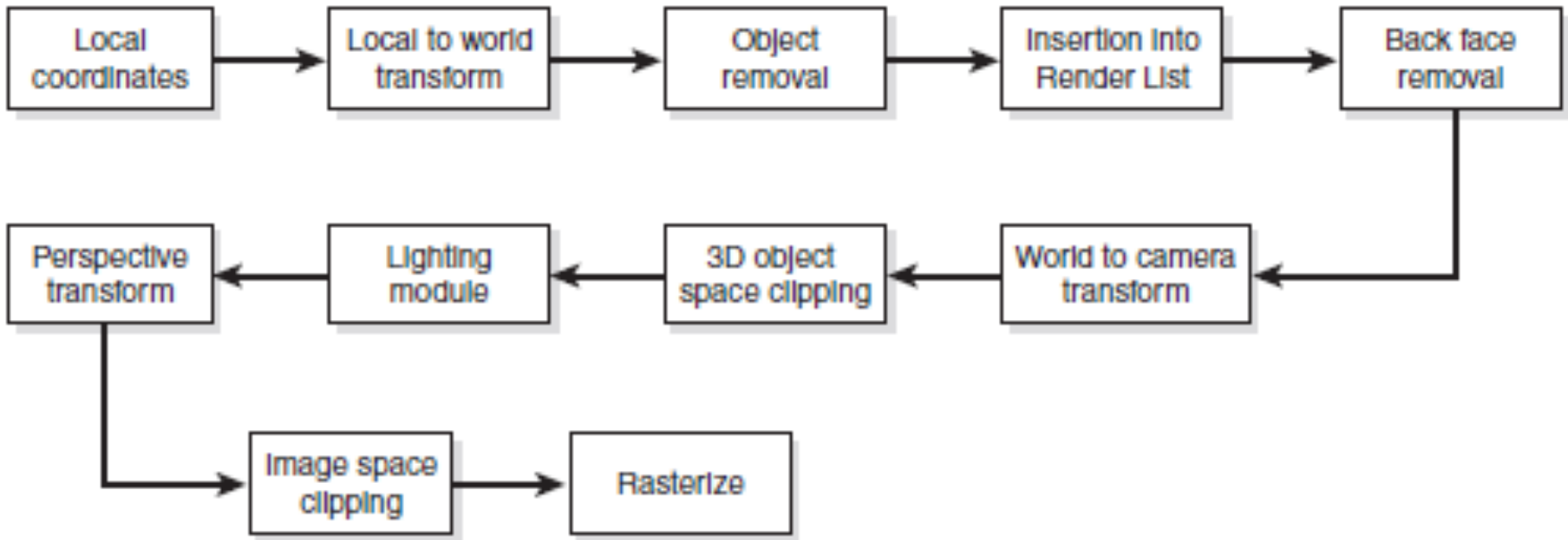
1000ms/sec / 300 FPS = 3.333.. ms per frame

Our goal: Frame times of between 16ms and 30ms

(1000ms/sec / 60 FPS = ~17ms) (1000ms/sec / 30 FPS = ~32ms)

http://www.mvps.org/directx/articles/fps_versus_frame_time.htm

The Transformation Pipeline



Clipping stages we are going to use:

1. Cull objects (this is our Quake 3 BSP level culling)
2. Remove back faces
3. Clip all polygons to viewing frustum
    - True clipping to the near plane
    - Trivial rejection against top/bottom, left/right
4. 2D clipping during rasterization

...it's 106 miles to Chicago, we got a full tank of gas, half a pack of cigarettes, it's dark and we're wearing sunglasses...

Fundamental Transforms

Translation

$$\begin{bmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale

$$\begin{bmatrix} X & 0 & 0 & 0 \\ 0 & Y & 0 & 0 \\ 0 & 0 & Z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation

x-axis

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

y-axis

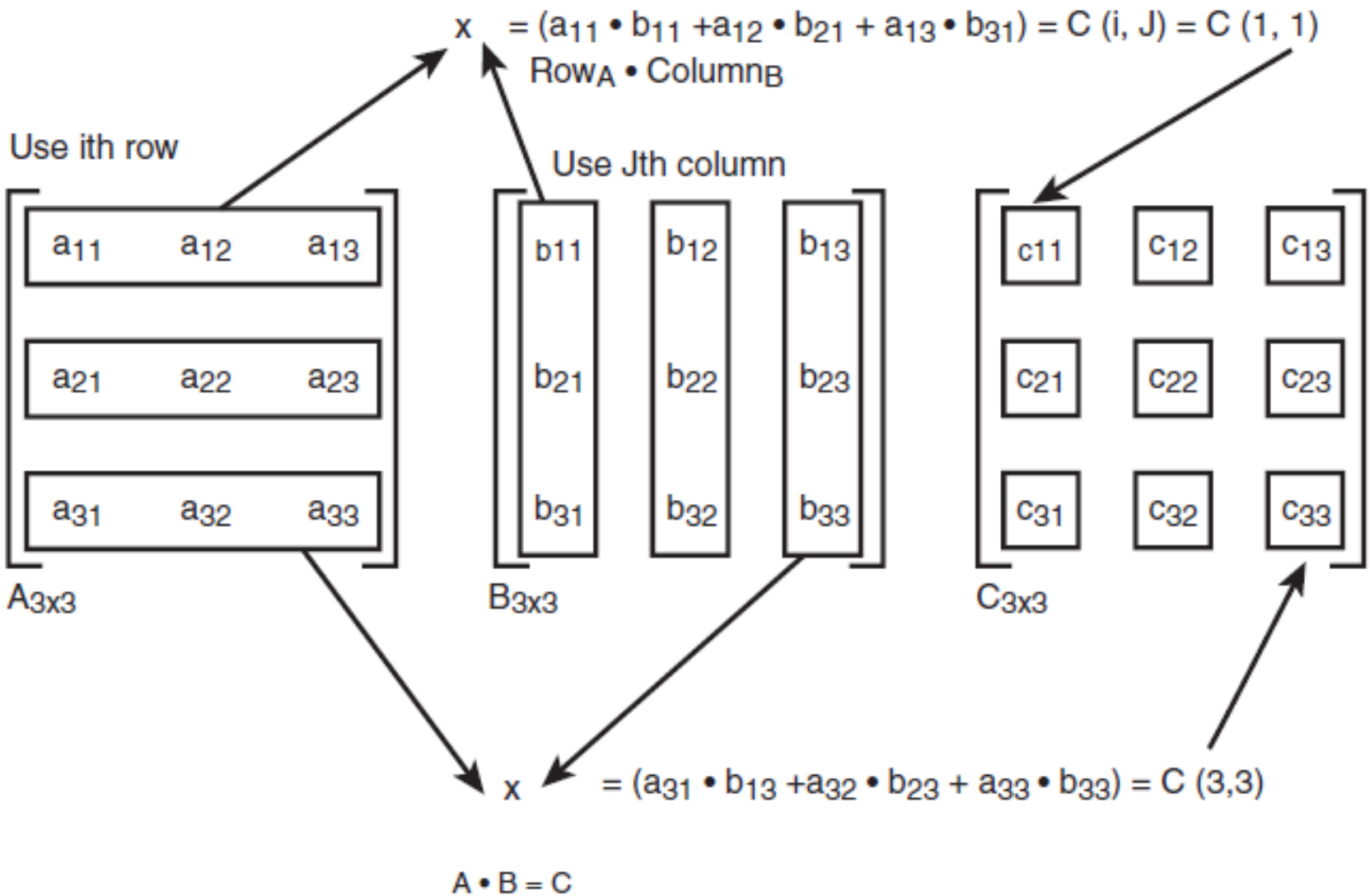
$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

z-axis

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

we will add more as we need them...

Matrix Multiplication



Homogeneous Coordinates

Normally we think of a coordinate in 3D space as:

$$[x, y, z]$$

but in order to perform vector * 4x4 matrix operations we need an additional term: w

We can convert a homogeneous coordinate back to a normal coordinate:

$$[x / w, y / w, z / w]$$

w is used to represent the concept of our values going towards infinity

consider the linear equation:

$$a * x + b * y + c = 0$$

this equation can be said to be linear and with a degree of 1

however, two of the terms have a degree of 1, while the constant term has a degree of 0

In order to make this equation homogeneous (all terms of the same degree) we can use w

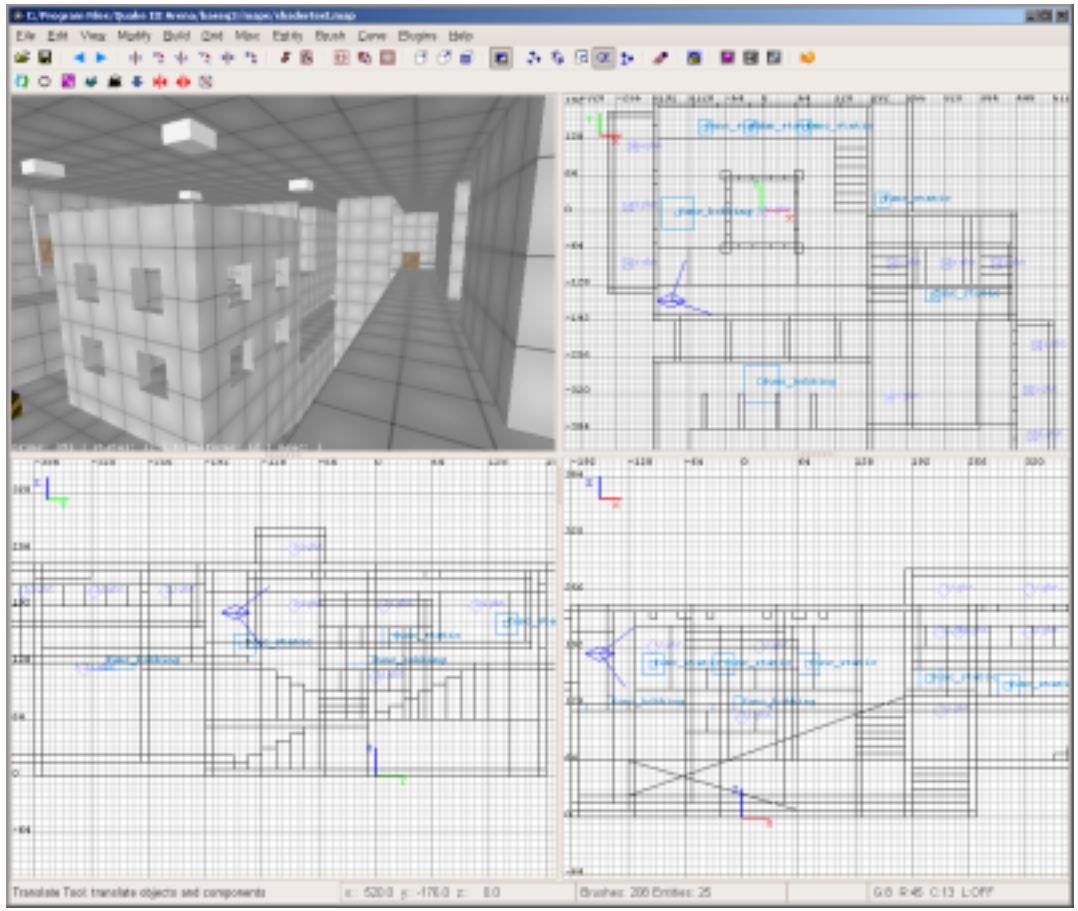
$$a * x / w + b * y / w + c = 0$$

Dividing through:

$$a * x + b * y + c * w = 0$$

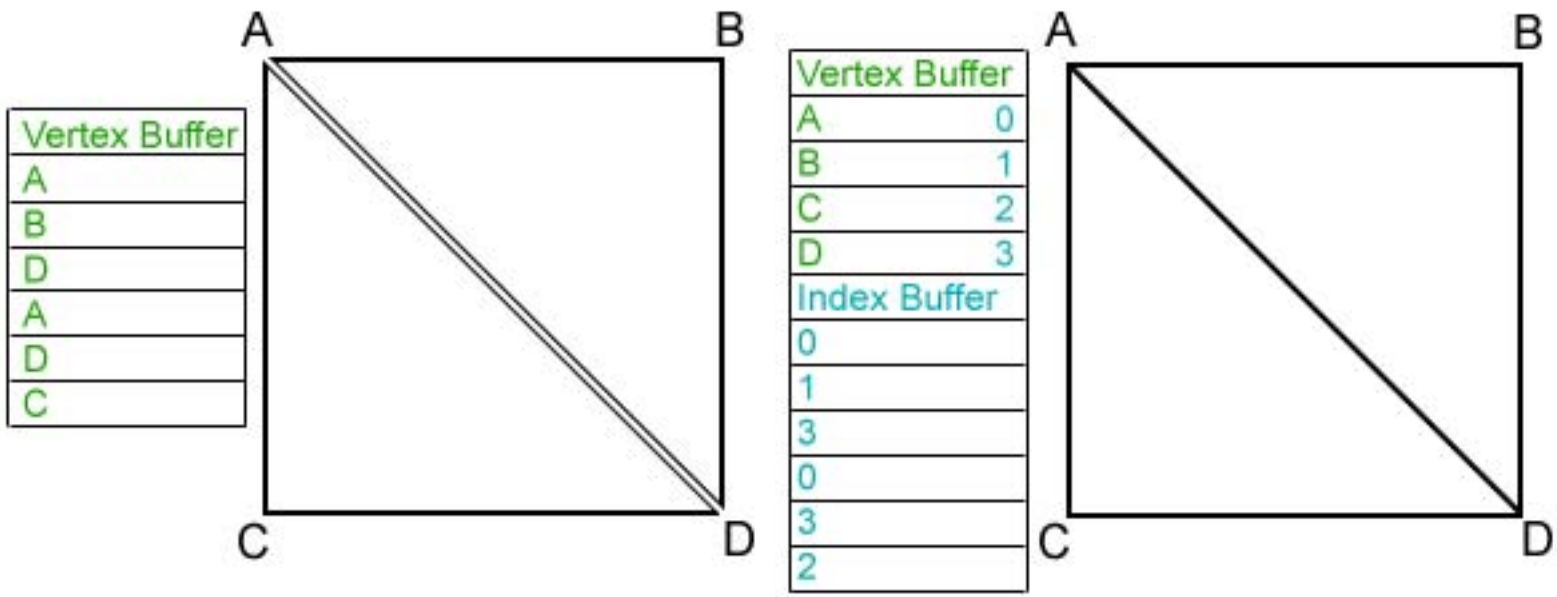
Take away point: Using homogeneous coordinates guarantees all our transforms will work

Local Coordinates



gtkRadiant (http://www.qeradiant.com/cgi-bin/trac.cgi)

Indexed Vertices



Vertex and Index Buffers, Chad Vernon
http://www.chadvernon.com/blog/tutorials/managed-directx-2/vertex-and-index-buffers/

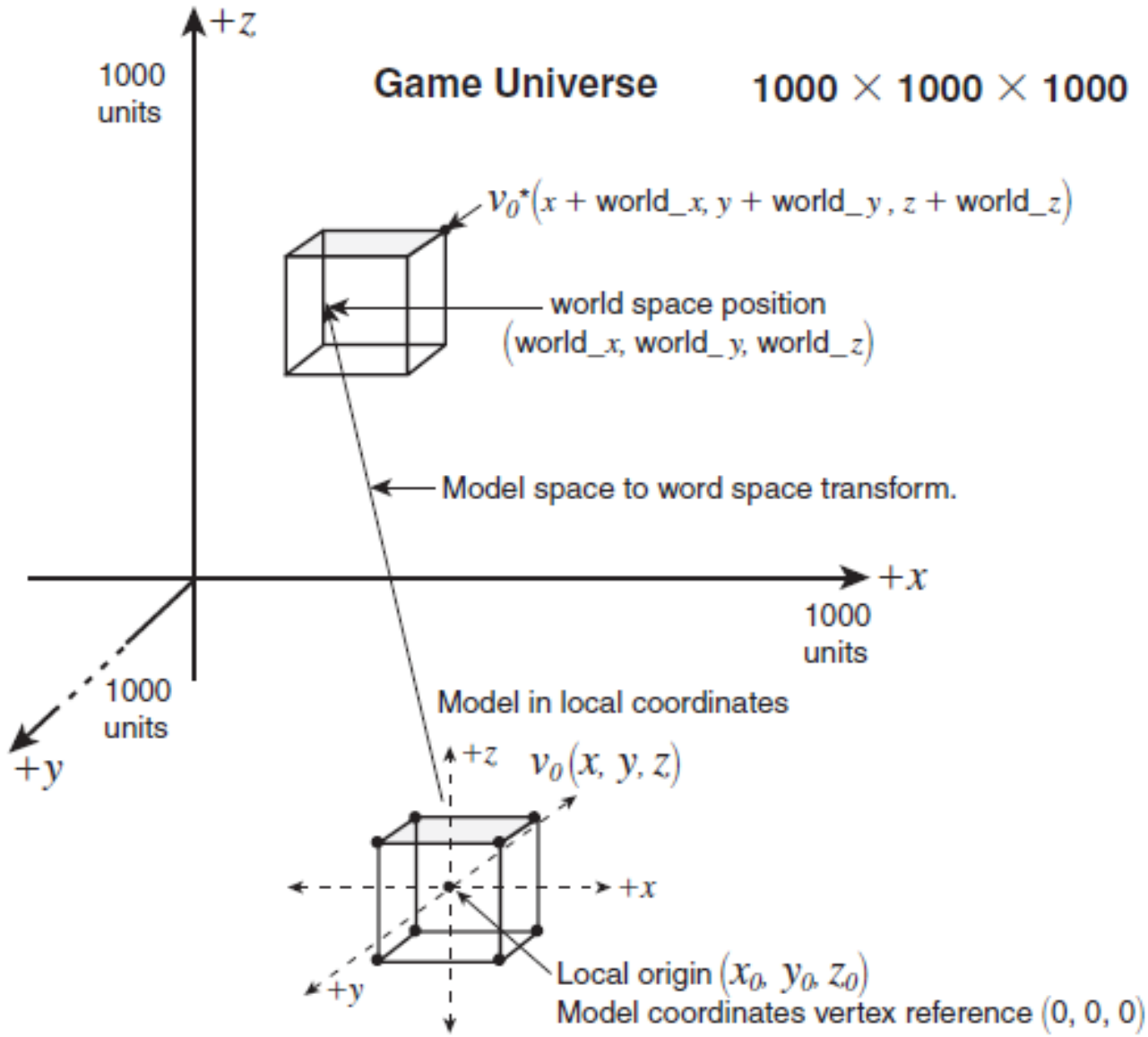
.obj Model Files

```
v 0.296502 -0.907931 0.450151
v 0.315114 -0.913622 0.435867
v 0.324517 -0.920404 0.443869
...

f 256/258/259 259/258/261 260/262/263
f 257/256/259 242/244/243 282/249/253
f 258/260/261 286/268/253 293/292/295
...

vt 0.0534429 0.20767 -0.0290818
vt 0.159879 -0.170688 -0.373284
vt 0.146713 -0.17011 -0.374094
...

vn -0.167471 -0.490408 -0.38523
vn -0.175871 -0.471666 -0.385346
vn -0.184067 -0.452822 -0.385423
...
```

Collada Model Files

```
<float_array id="pCubeShape1-positions-array" count="24">-
0.5 -0.5 0.5 0.5 -0.5 0.5 -0.5 0.5 ...

<float_array id="pCubeShape1-map1-array" count="28">0.375 0
0.625 0 0.375 0.25 0.625 0.25 ...

<polylist material="initialShadingGroup" count="6">
          <input semantic="VERTEX" source="#pCubeShape1-
vertices" offset="0"/>
          <input semantic="NORMAL" source="#pCubeShape1-
normals" offset="1"/>
          <input semantic="TEXCOORD" source="#pCubeShape1-
map1" offset="2" set="0"/>
          <vcount>4 4 4 4 4 4</vcount>
          <p>0 0 0 1 1 1 3 2 3 2 3 2 2 4 2 3 5 3 5 6 5 4 7
4 4 8 4 5 9 5 7 10 7 6 11 6 6 12 6 7 13 7 1 14 9 0 15 8 1
16 1 7 17 10 5 18 11 3 19 3 6 20 12 0 21 0 2 22 2 4 23
13</p>
        </polylist>
```

Local to World Transform

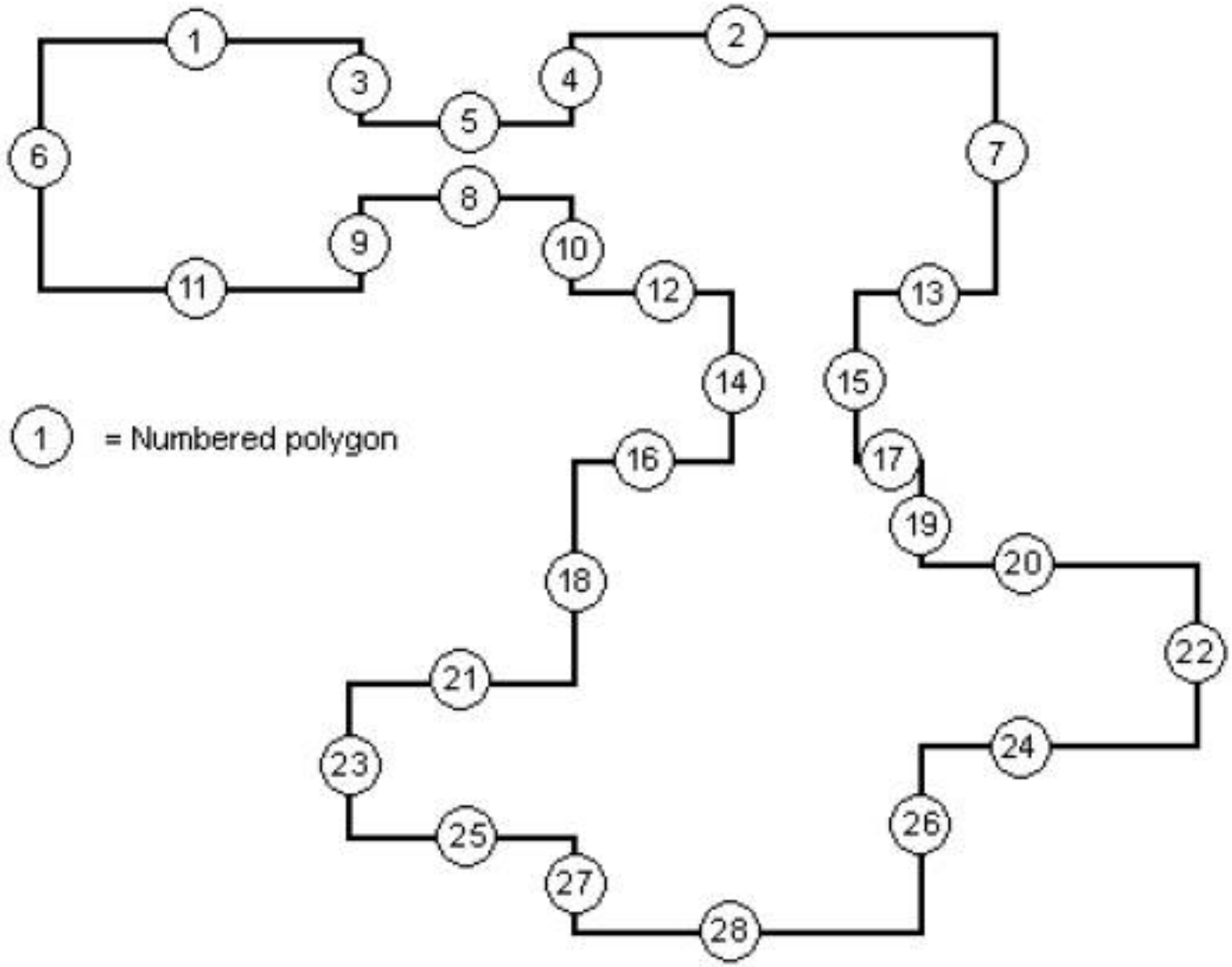


Object Removal

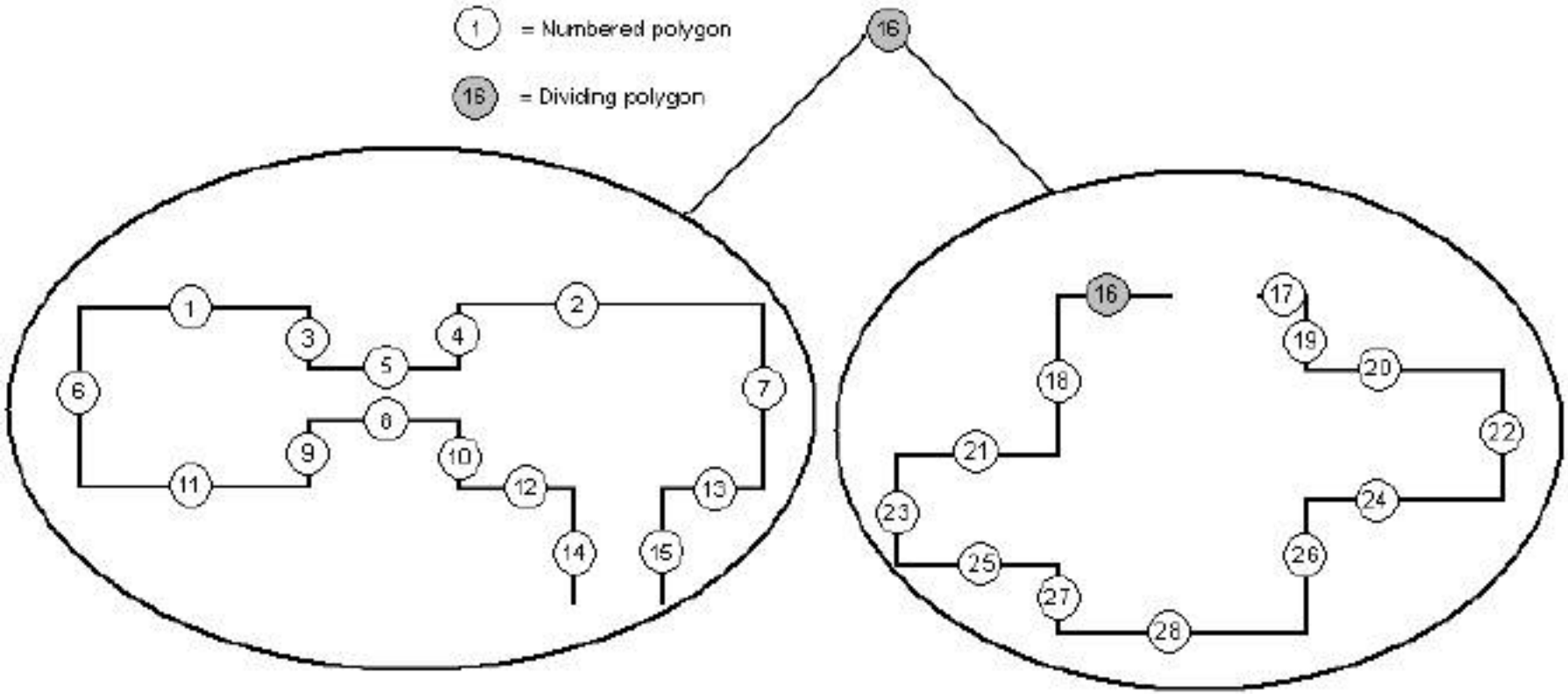
Quake 3 .bsp files (and many other game level files) use binary space partitioning (BSP) tree to divide up the level

Using this BSP tree data structure we can accelerate rendering by:

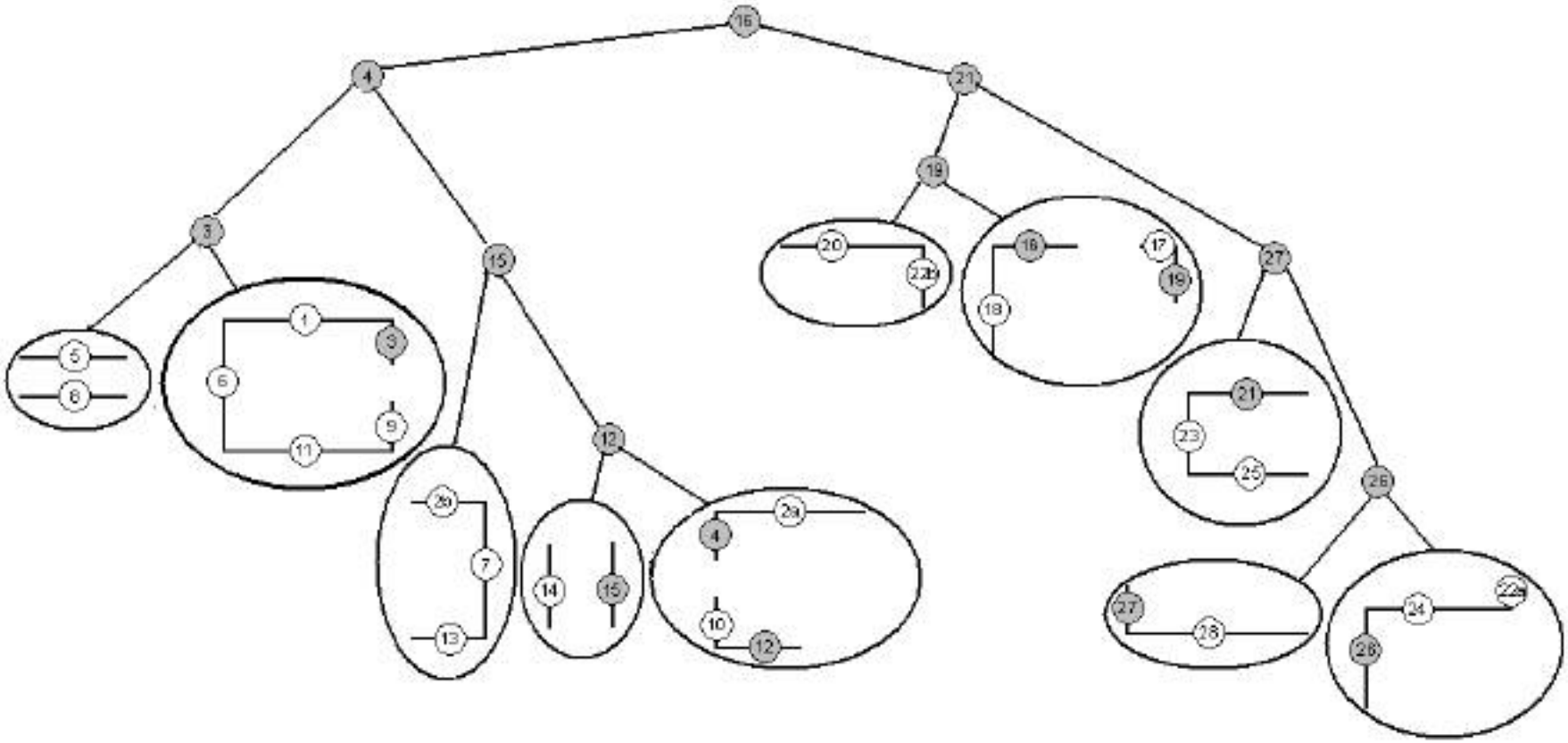
- Frustum culling
  - Determine if nodes (as defined by their AABB) are inside the frustum
  - Furthermore, because we are using a tree we can quickly 'prune' branches reducing the number of tests required
- Visibility Bitset
  - Each node has a bitset which can be compared to any other node to check for potential visibility



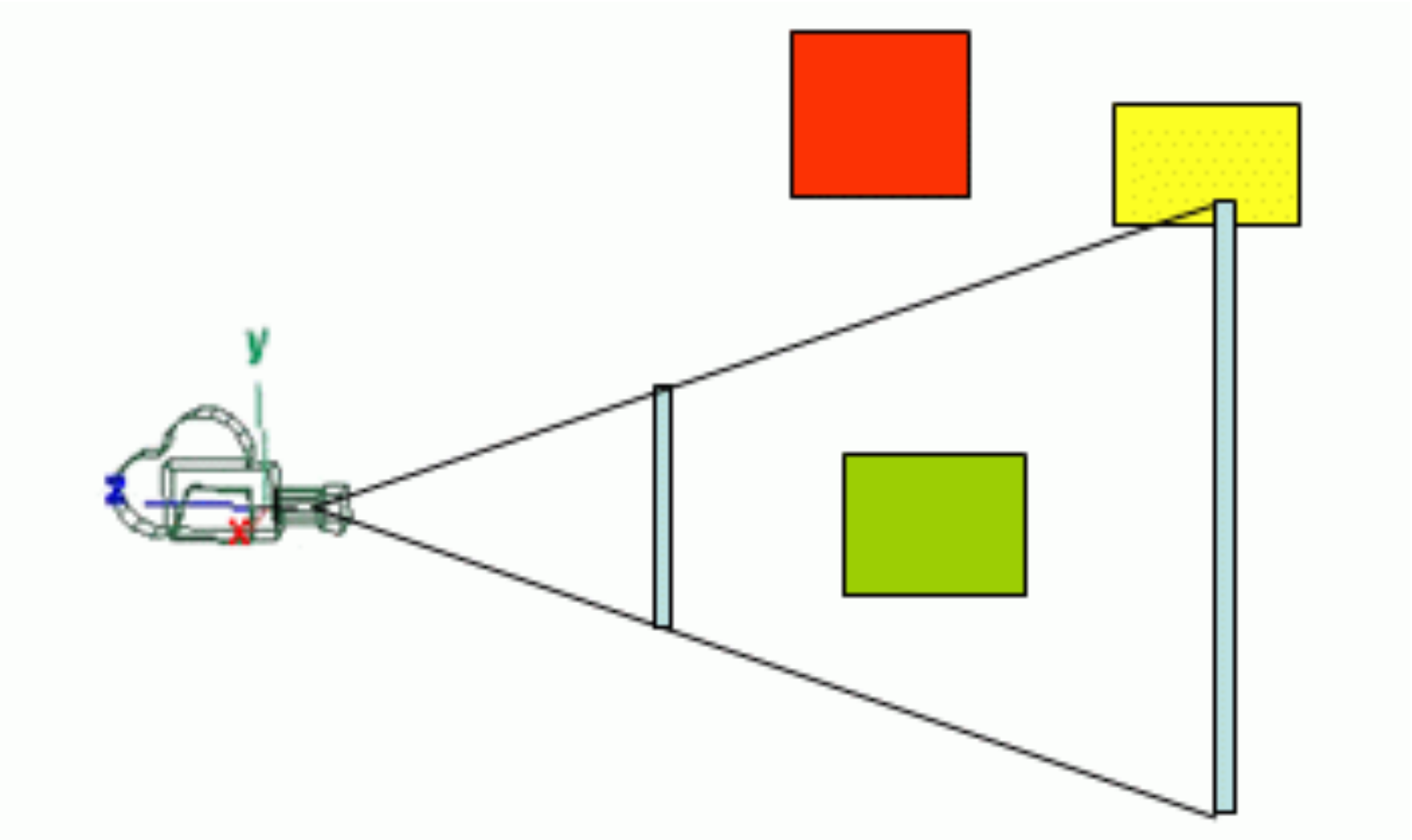
BSP Trees: Theory and Implementation (http://www.devmaster.net/articles/bsp-trees/)



We make the partitions along the polygons that split the fewest number of polygons. This rule gives the best relation between the sizes of the two resulting sets.



Each node (even those which are 'empty') has an AABB so we can test parent nodes and potentially eliminate the need for tests on child nodes.



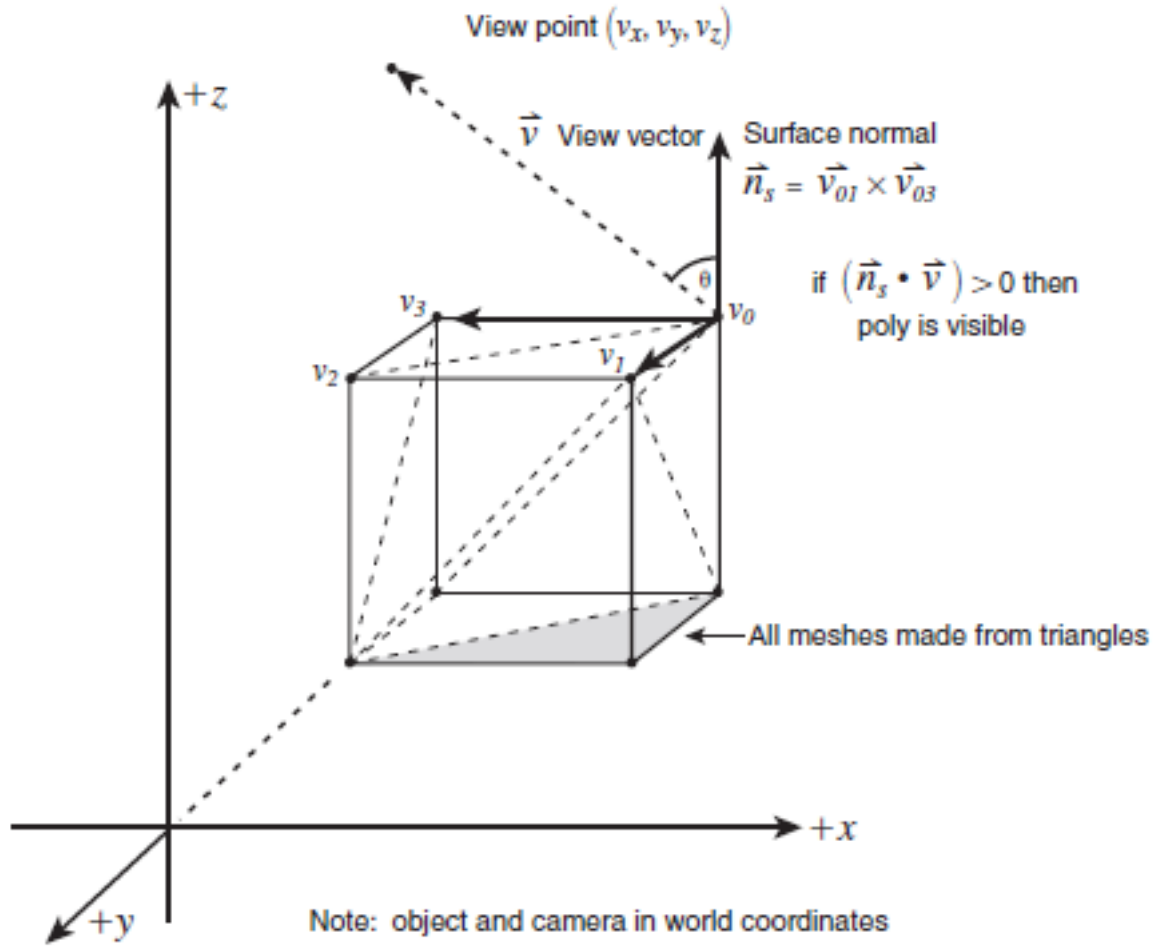
View Frustum Culling http://www.lighthouse3d.com/opengl/viewfrustum/index.php?gatest2

Code examples in C++..?!?!1

Cluster Visibility

```
inline int Q3BSP::IsClusterVisible(int current, int test)
{
    if(!m_clusters.pBitsets || current < 0) return 1;

    char visSet =
        m_clusters.pBitsets[(current*m_clusters.bytesPerCluster) + (test /
8)];

    int result = visSet & (1 << ((test) & 7));

    return ( result );
}
```

Back-face Culling

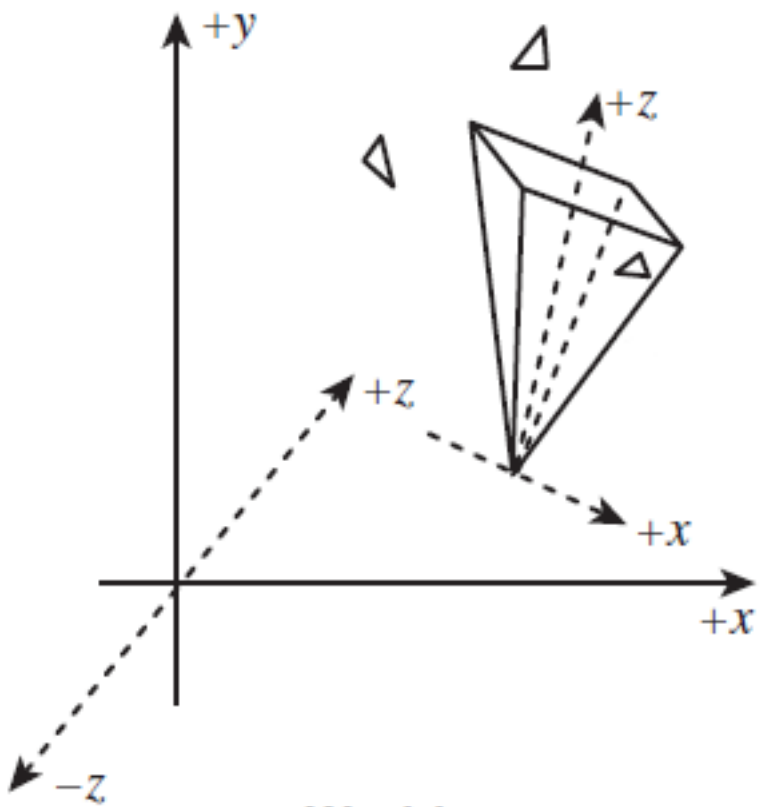


World to Camera Transform

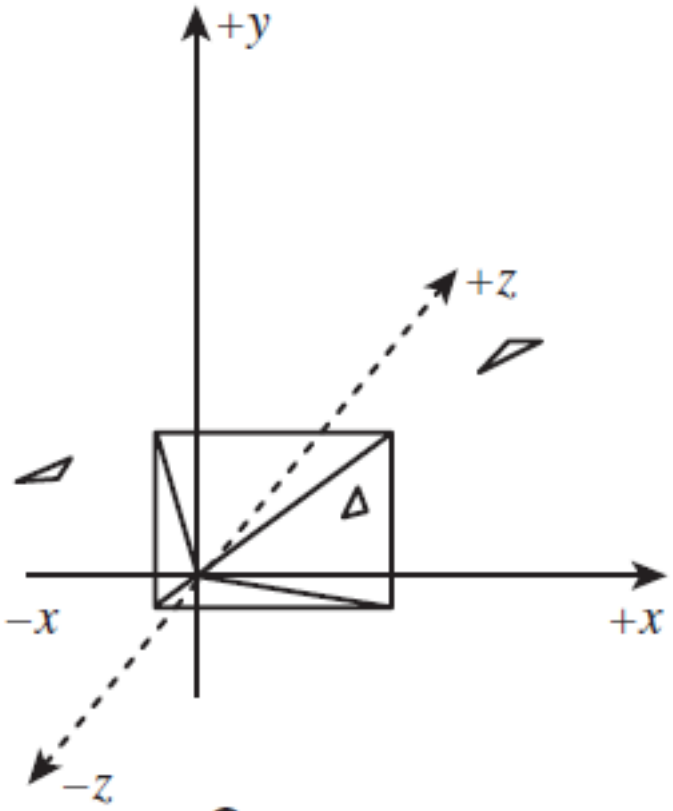
Transform vertices according to the rotation and translation of the camera

Sometimes easier to think of this in terms of a camera position, look vector and up vector

```
Mat4x4 CreateLookAtMatrix(Vec3 &cameraPosition, Vec3 &cameraTarget, Vec3 &upVector)
{
    Mat4x4 rMat;

    Vec3 zaxis = (cameraTarget - cameraPosition);
    normalizeSelfFAST(zaxis);

    Vec3 xaxis = cross(upVector, zaxis);
    normalizeSelfFAST(xaxis);

    Vec3 yaxis = cross(zaxis, xaxis);
    normalizeSelfFAST(yaxis);

    rMat.m[0][0] = xaxis.x; rMat.m[0][1] = yaxis.x; rMat.m[0][2] = zaxis.x; rMat.m[0][3] = 0;
    rMat.m[1][0] = xaxis.y; rMat.m[1][1] = yaxis.y; rMat.m[1][2] = zaxis.y; rMat.m[1][3] = 0;
    rMat.m[2][0] = xaxis.z; rMat.m[2][1] = yaxis.z; rMat.m[2][2] = zaxis.z; rMat.m[2][3] = 0;

    rMat.m[3][0] = -dot(xaxis, cameraPosition);
    rMat.m[3][1] = -dot(yaxis, cameraPosition);
    rMat.m[3][2] = -dot(zaxis, cameraPosition);
    rMat.m[3][3] = 1;

    return rMat;
}
```

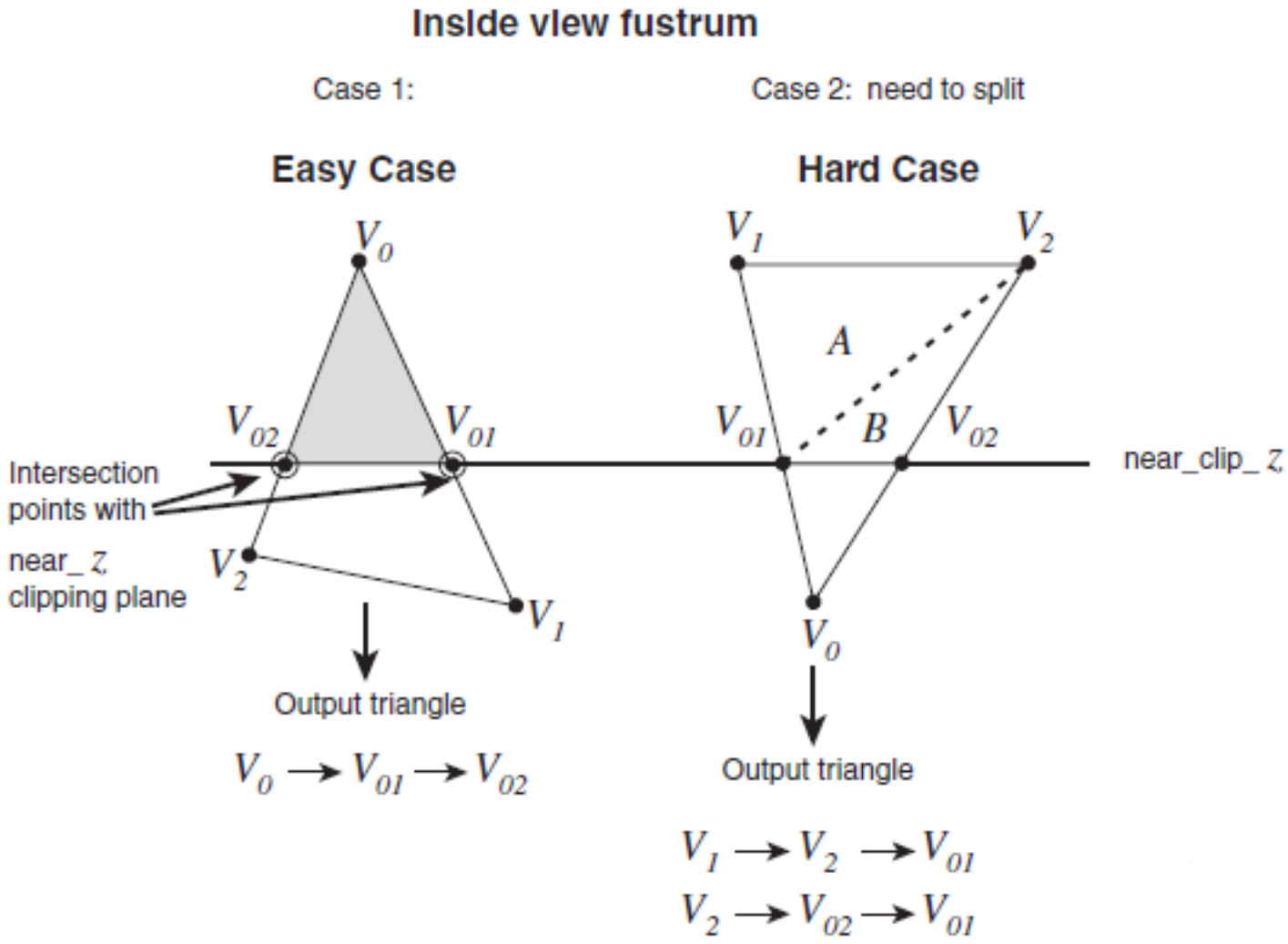


World space

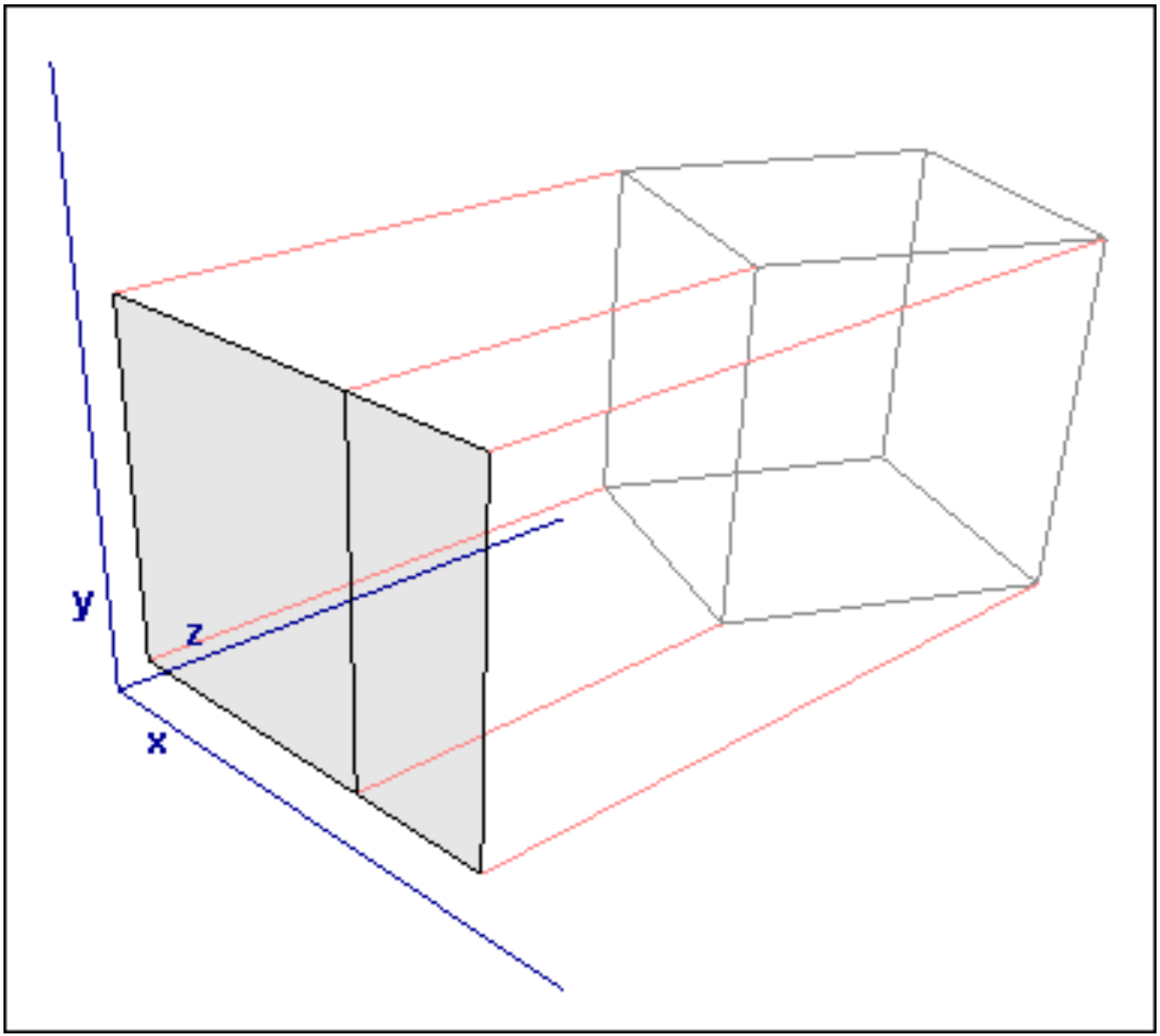


Camera space

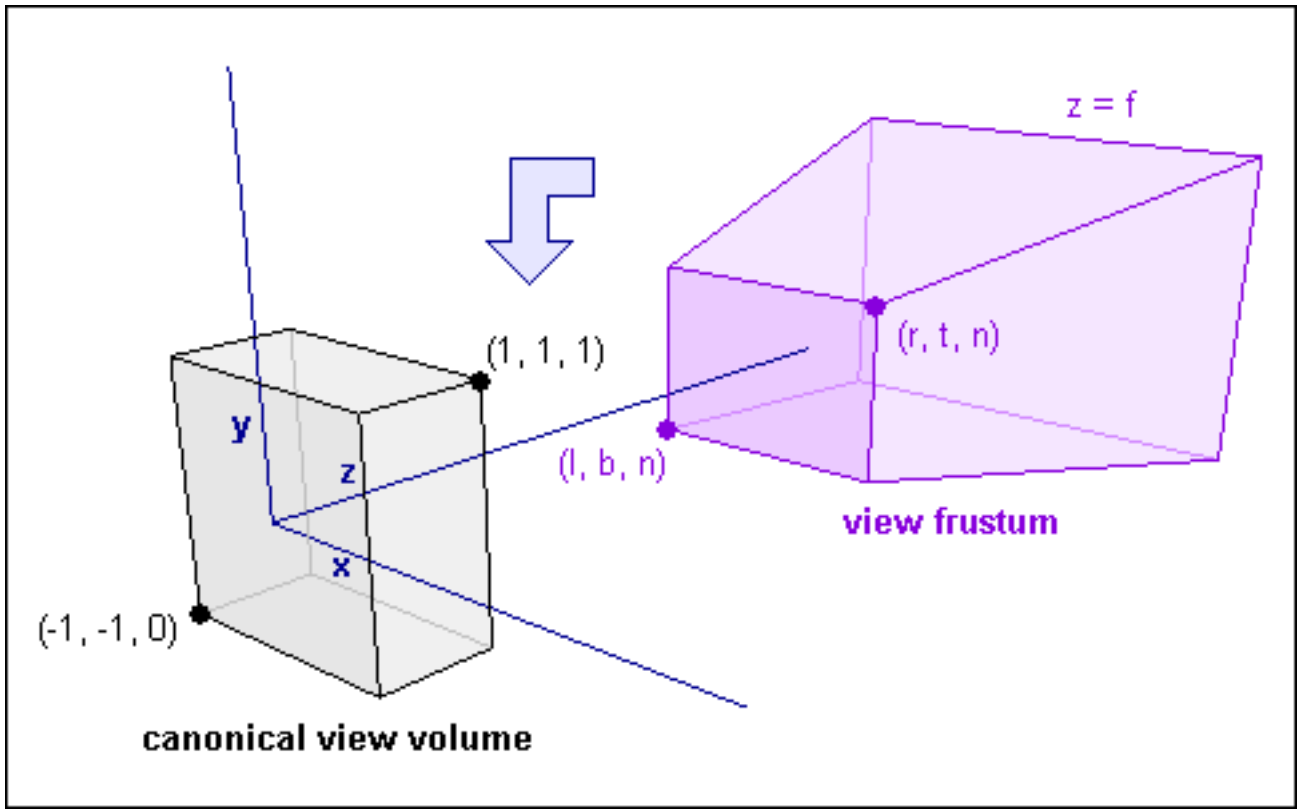
3D Object Space Clipping



Projection to a plane by discarding z-coordinates



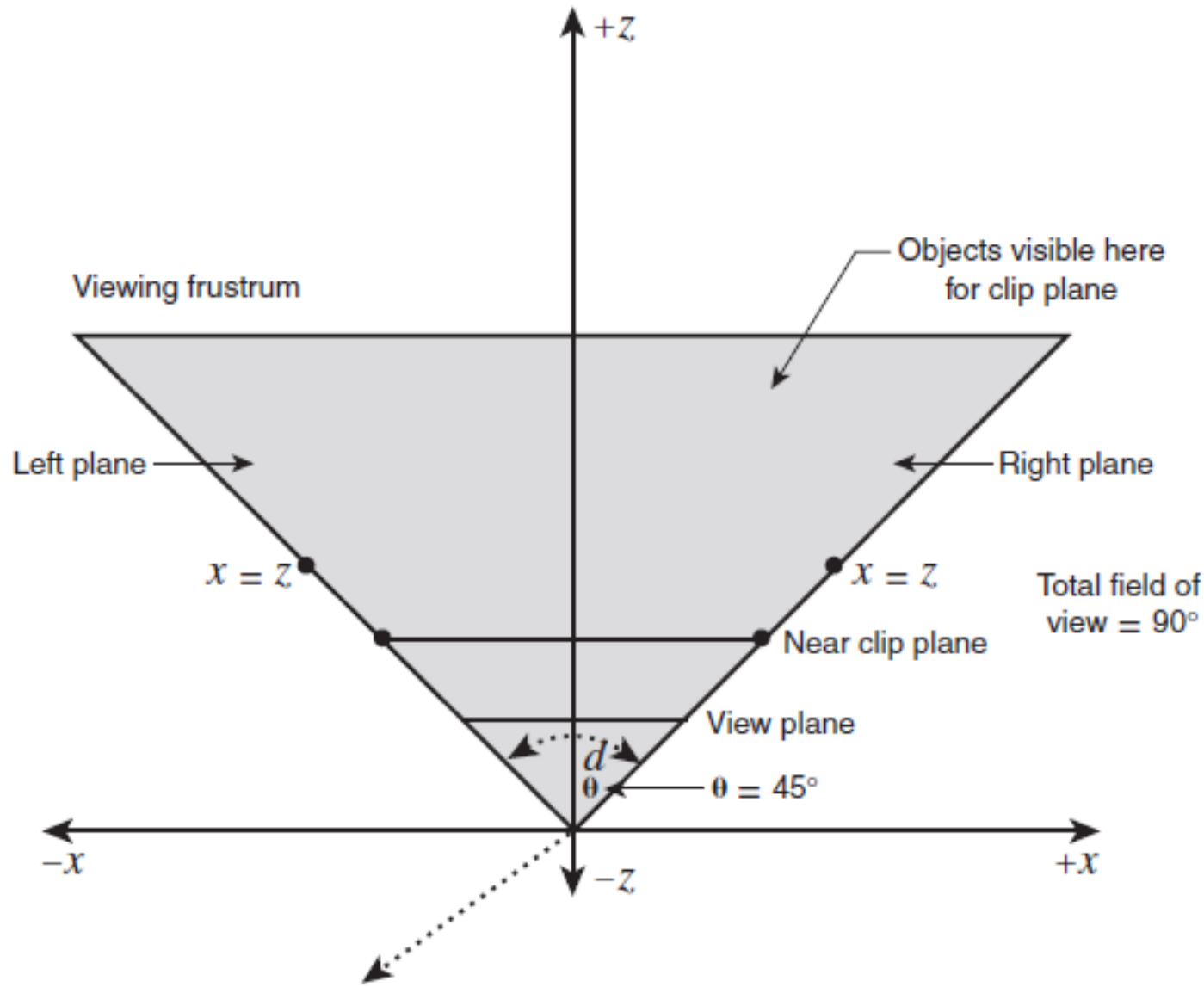
Perspective Transform



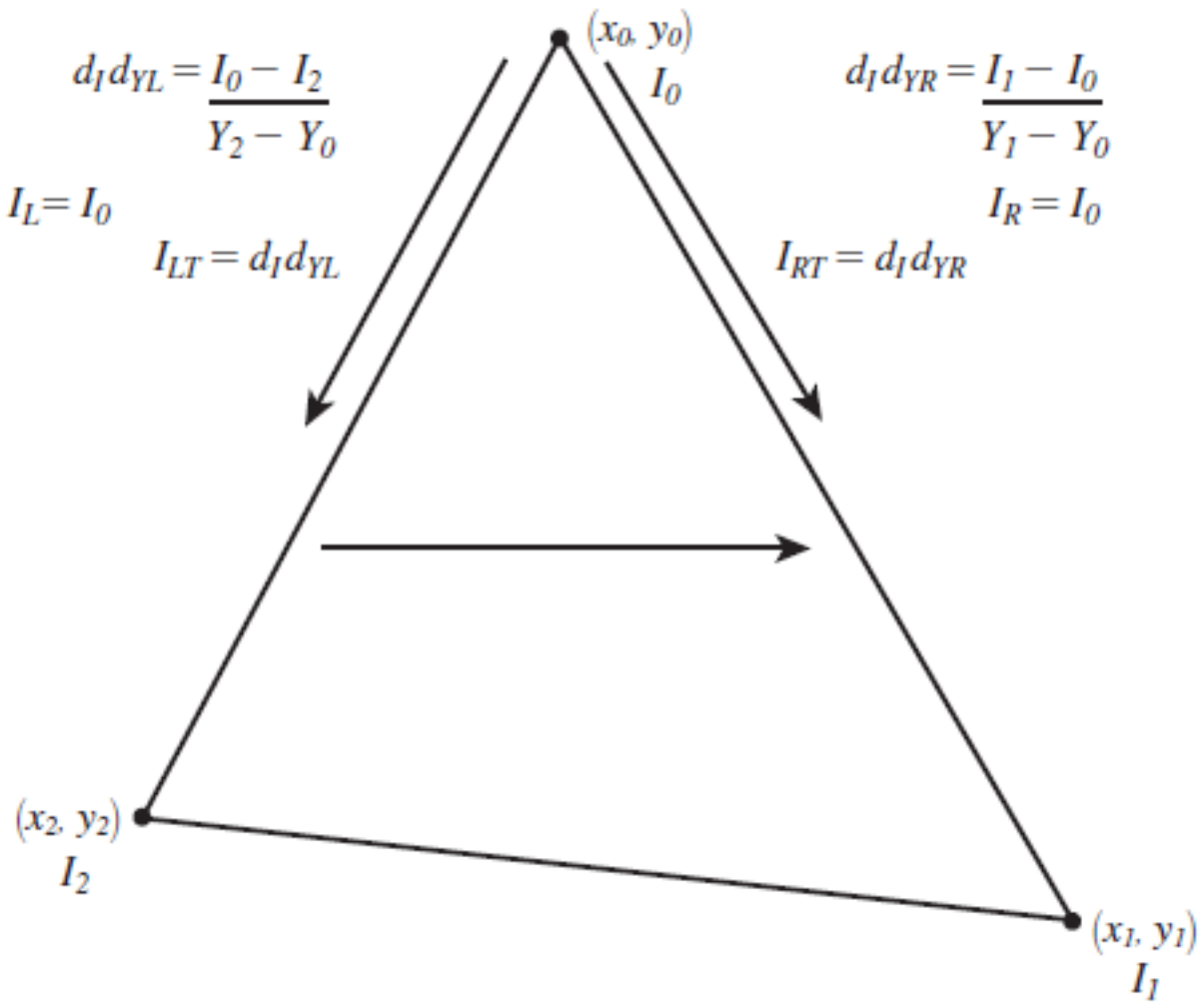
Projection Matrix

$$M_{proj} = \begin{pmatrix} \frac{2 * Zn}{Sw} & 0 & 0 & 0 \\ 0 & \frac{2 * Zn}{Sh} & 0 & 0 \\ 0 & 0 & \frac{Zf}{Zf - Zn} & 1 \\ 0 & 0 & \frac{-Zf * Zn}{Zf - Zn} & 0 \end{pmatrix}$$

Then divide by w to collapse the view frustum

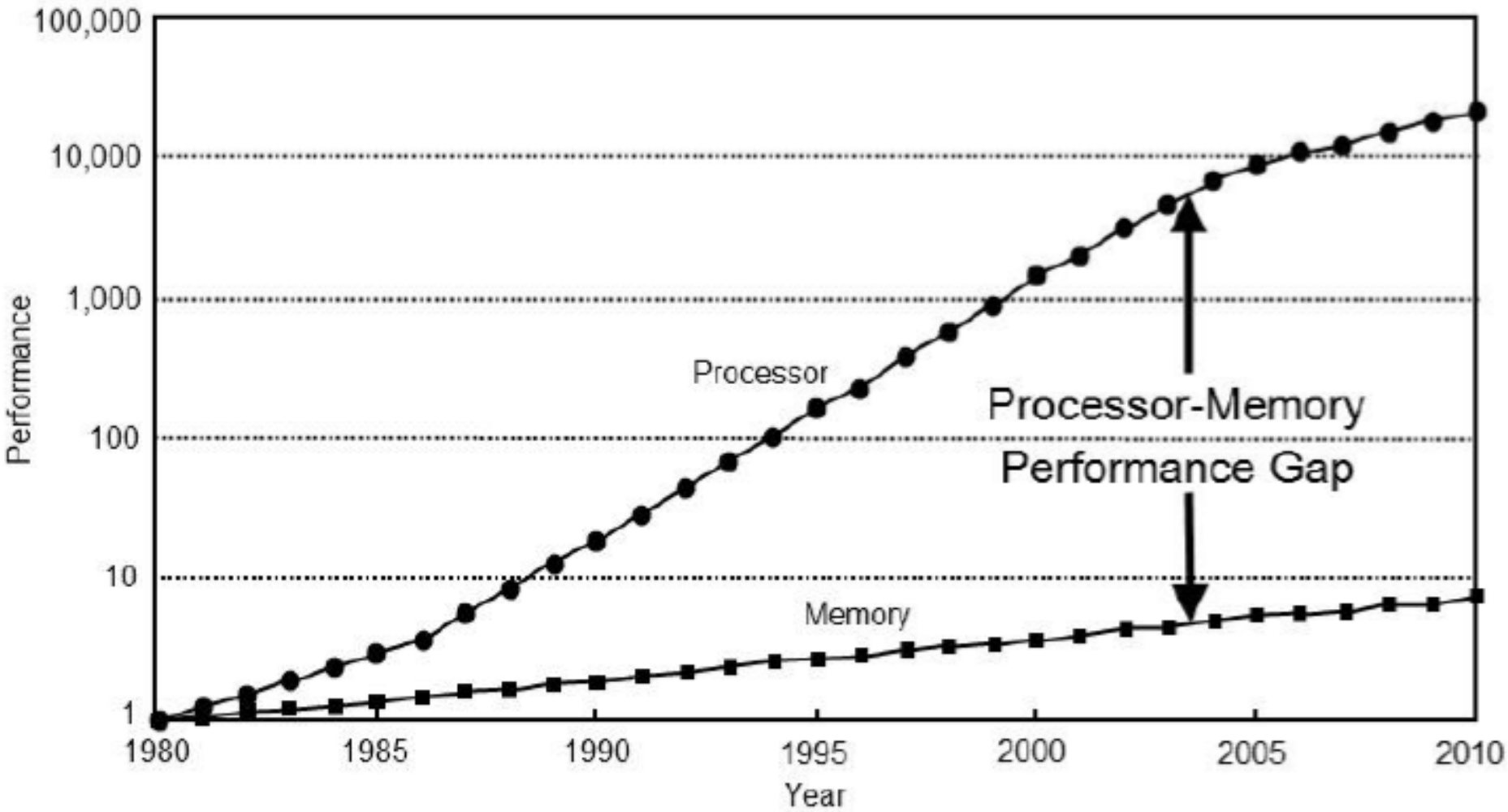

+z
Viewing frustrum
Objects visible here
for clip plane
Left plane
Right plane
x = z
x = z
Total field of
view = 90°
Near clip plane
View plane
d
θ
θ = 45°
−x
+x
−z

Rasterization



I_0, I_1, I_2 are the values we are interpolating, could be color, texture, z values etc.

A word about memory (before I forget)



Computer architecture: a quantitative approach 3rd Edition
by John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau

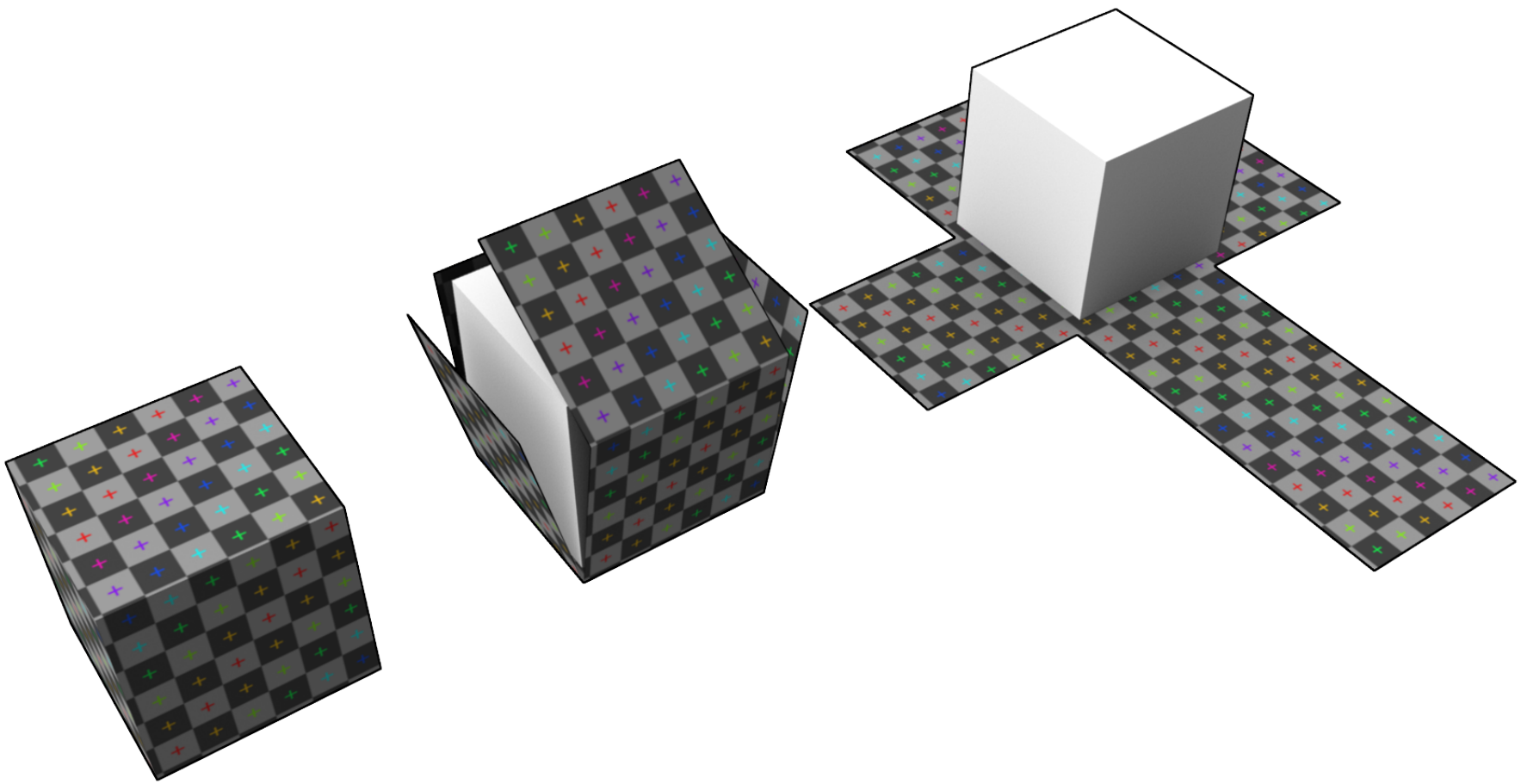
"Any memory access, especially random access like the affine bitmap rendering is the most expensive operation."

"In the end the conclusion remains: It is memory access time which kills us."

- Tinic Uro, Flash Player Engineer, February 2007

http://www.kaourantin.net/2007/02/limits-of-software-rendering.html

Texture Mapping



Convert UV Coordinates to Pixel Coordinates

simplest:

```
pixel_x = u * texture_width
pixel_y = v * texture_height
```

handle wrapping UVs:

```
pixel_u = (u - floor(u)) * texture_width
pixel_v = (v - floor(v)) * texture_height
```

Using SSE for faster float->int conversions and bitshifts for multiplications:

```
inline int ftoi_sse1(float f)
{
    // SSE1 instructions for float->int
    return _mm_cvtt_ss2si(_mm_load_ss(&f));
}
```

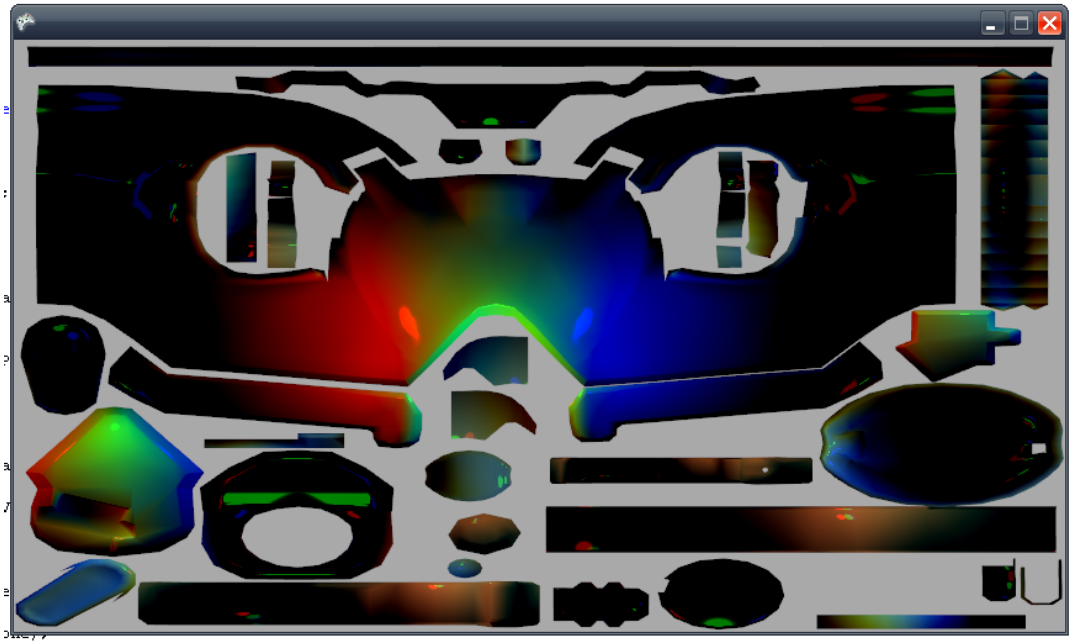
```
const int color = texture[ ( ftoi_sse1(diffuse_v) << 7 ) + ftoi_sse1(diffuse_u) ];
const int light = lightmap[ ( ftoi_sse1(lightmap_v) << 7 ) + ftoi_sse1(lightmap_u) ];
```

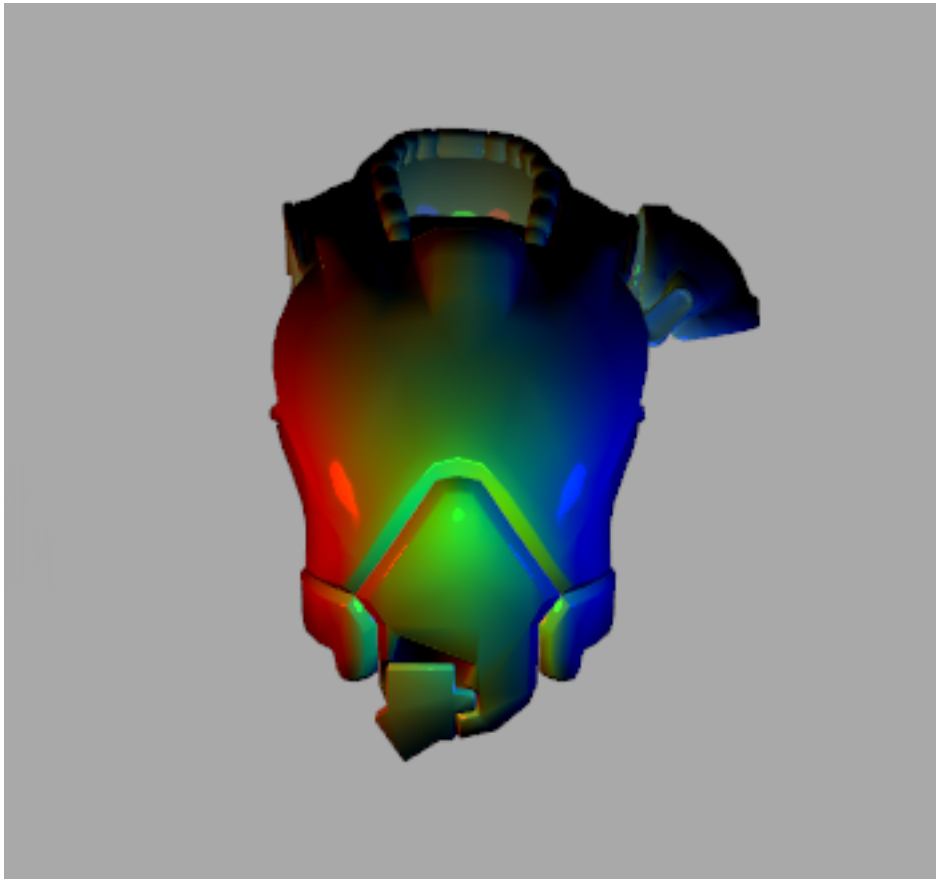
Extract color values:

```
const int color_r = ((color >> 16) & 0xFF);
const int color_g = ((color >> 8) & 0xFF);
const int color_b = (color & 0xFF);
```

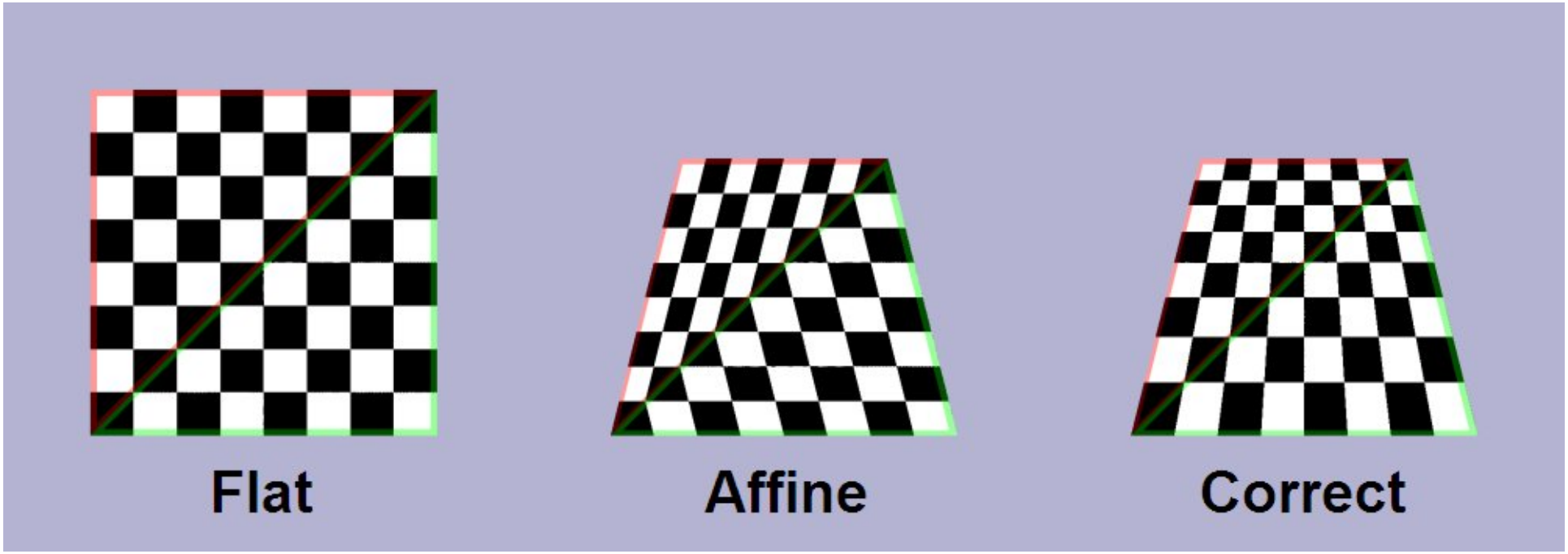
```
[   a    ][   r    ][   g    ][   b    ]
[ byte ][ byte ][ byte ][ byte ]
```

http://lab.polygonal.de/2007/05/10/bitwise-gems-fast-integer-math/





Perspective Correction



Linear interpolation only works for those things which are linear in screen space.

Therefore, we cannot linearly interpolate texture coordinates, z-coordinates or (even color intensities.)

However, 1/z is linear in screen space, so we can use that information to interpolate other data.

Just need to remember:

Interpolate z as 1/z, u as u/z and v as v/z

We can 'recover' the values as such:

```
(u/z) / (1/z) = u
```

Pro-tip (although bring your own grain(s) of salt):

Use reciprocals to avoid division(s)

Example:

```
r = r / 255.0f; // Requires 3 division operations
g = g / 255.0f;
b = b / 255.0f;
```

Or:

```
float reciprocal = 1.0f / 255.0f; // 1 division
r = r * reciprocal; // 3 multiplications
g = g * reciprocal;
b = b * reciprocal;
```

Very rough approximate numbers on how many cycles certain operations are going to take (these values vary by CPU):

1	addition, subtraction, comparison
2	fabs
3	abs
4	multiplication
10	division, modulus
10	fsqrtf (approximation)
50	exp
60	sin, cos, tan
80	asin, acos, atan
100	pow

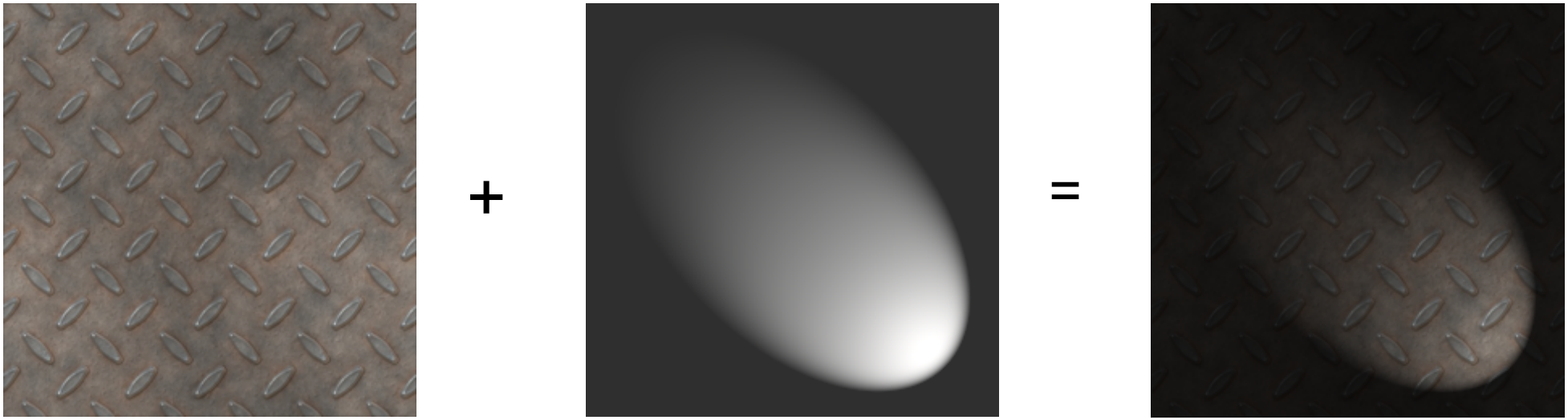
If you have stalled the CPU (with cache misses, etc) then you probably already have bigger problems! (remember our chart showing the relative CPU vs. memory performance)

If you are really into this kind of stuff, check out this thought provoking presentation:

"Pitfalls of Object Oriented Programming"
by Tony Albrecht

http://research.scee.net/files/presentations/gcapaustralia09/Pitfalls_of_Object_Oriented_Programming_GCAP_09.pdf

Light Mapping

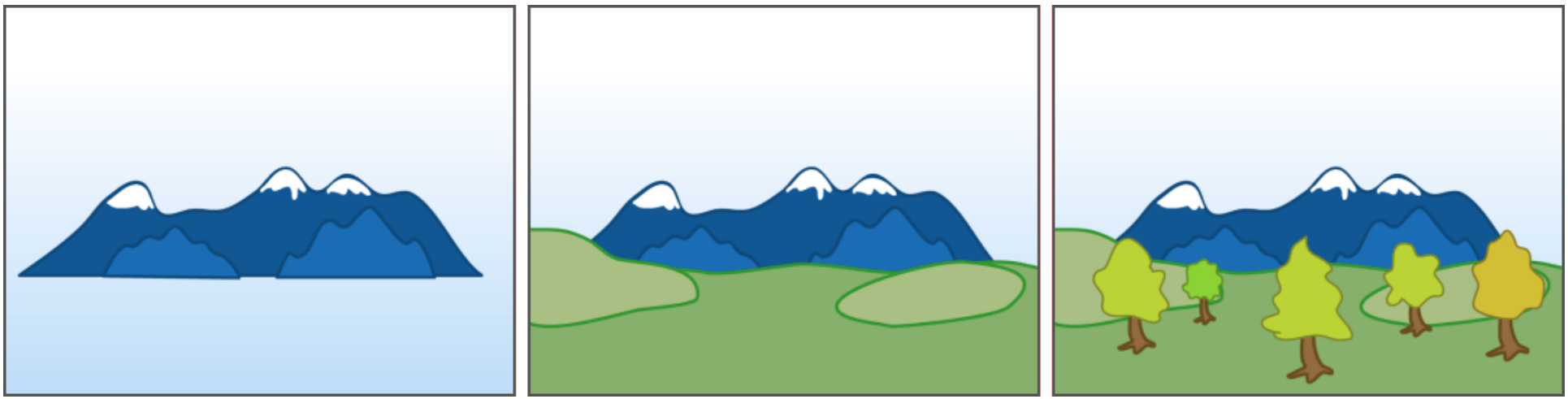


RGB Color (diffuse texture) * Luminance (normally a 0.0 - 1.0 value) = Lit Pixel Value

We can also handle integer light map values:

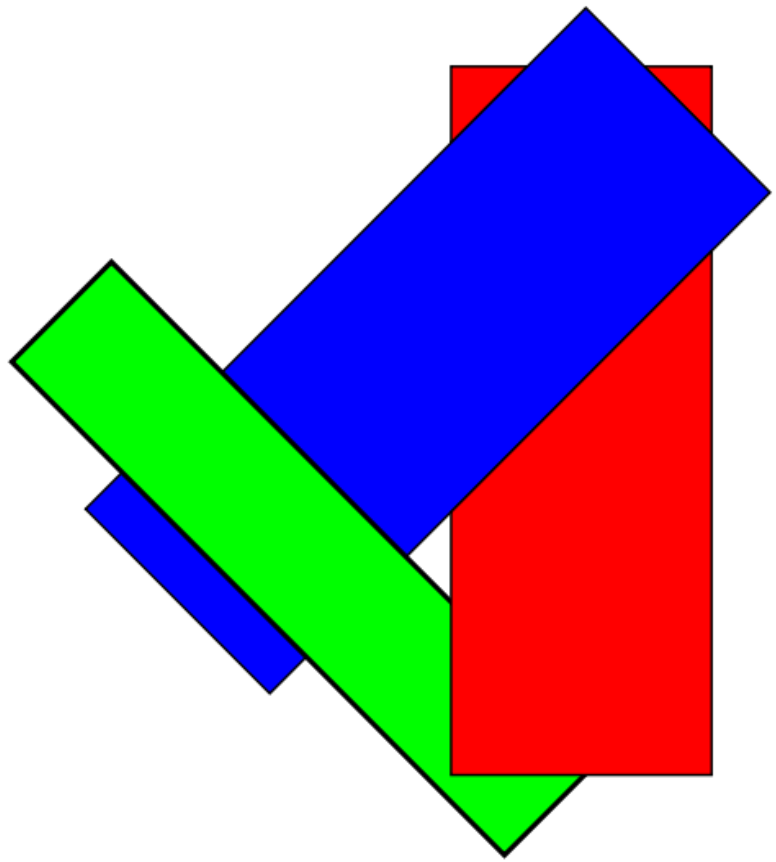
```
const int mix_r = (color_r * light_r) >> 8;
const int mix_g = (color_g * light_g) >> 8;
const int mix_b = (color_b * light_b) >> 8;
```

Painter's Algorithm



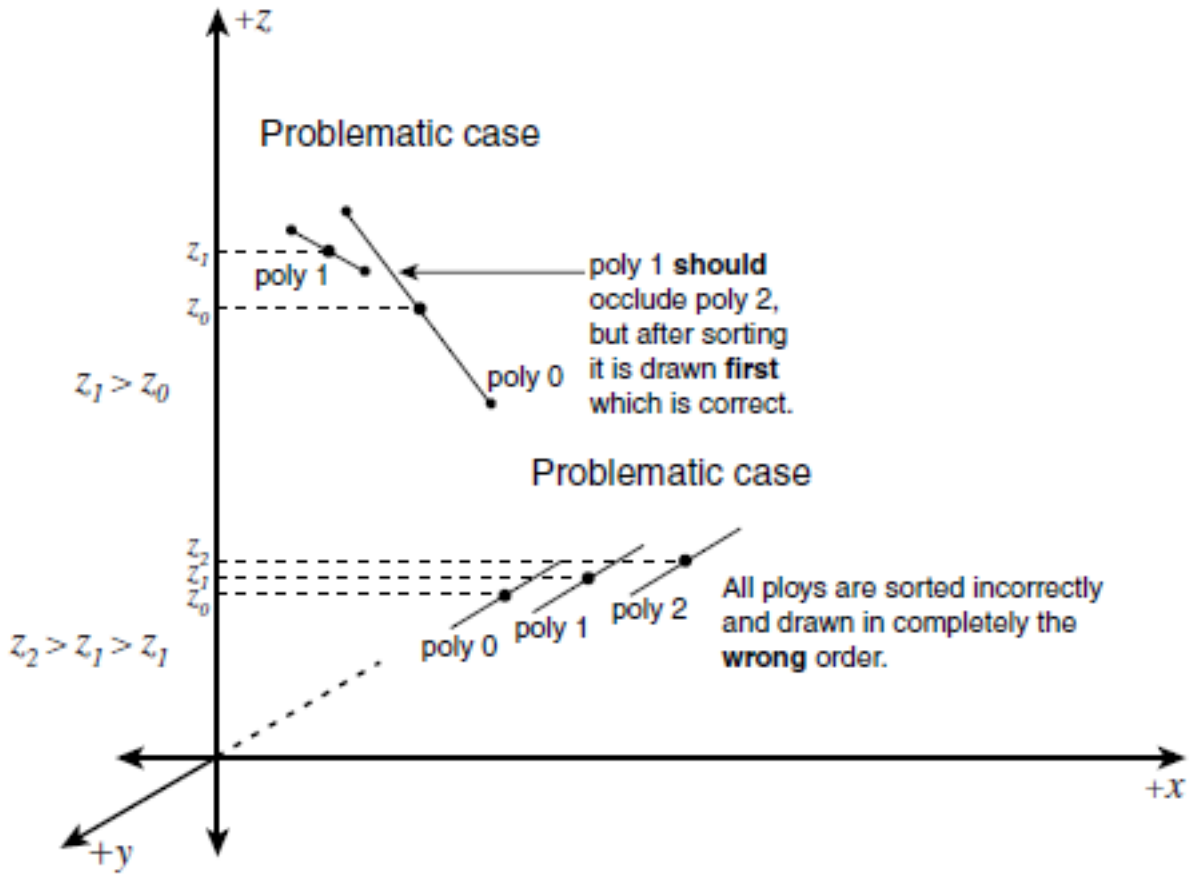
Render objects from back to front

Painter's Algorithm: Issues

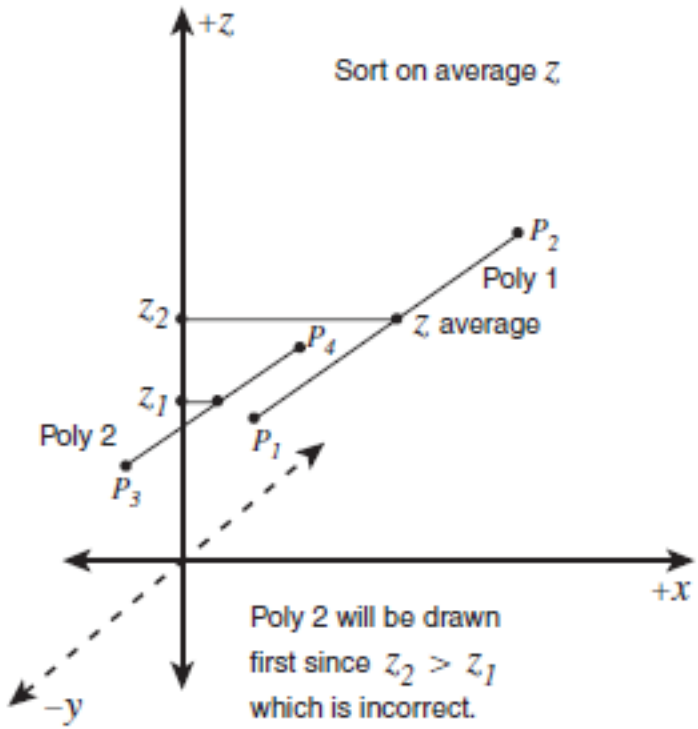


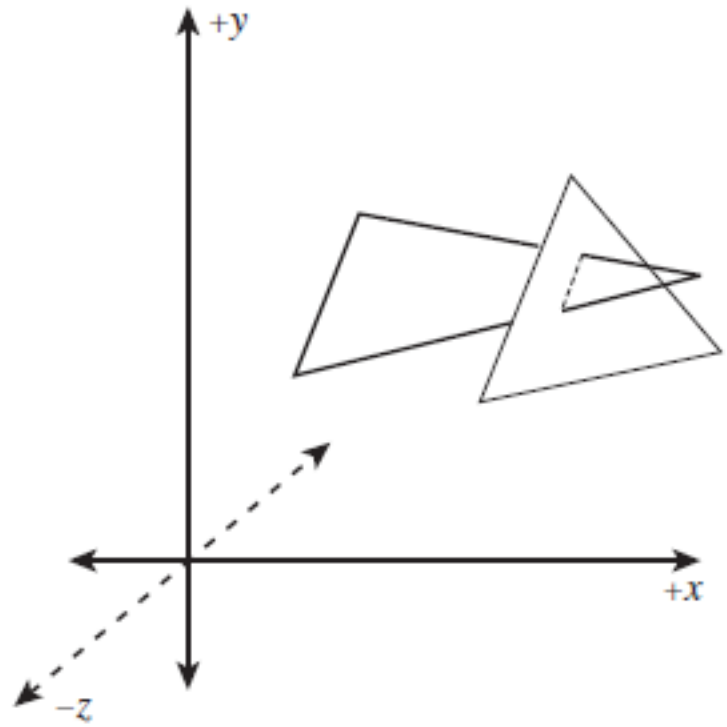
Which polygon is in front?

Painter's Algorithm: Trouble Cases



Painter's Algorithm: Trouble Cases





Z-Buffer (to the rescue!)

```
For each polygon in the rendering list begin

    1. Rasterize the polygon and generate xi, yi, zi
    for each value that the polygon's projection on
    the screen would generate during the rasterization.

    2. if (zi < zbuffer[xi, yi]) then write zi into the Z
    buffer,zbuffer[xi, zi] = zi, and then
    write the pixel to the screen, Plot(xi,yi).

end for
```

Z-Buffer (to the rescue!)