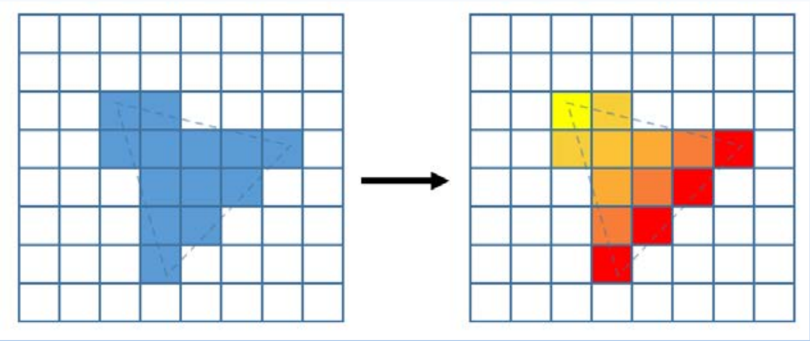
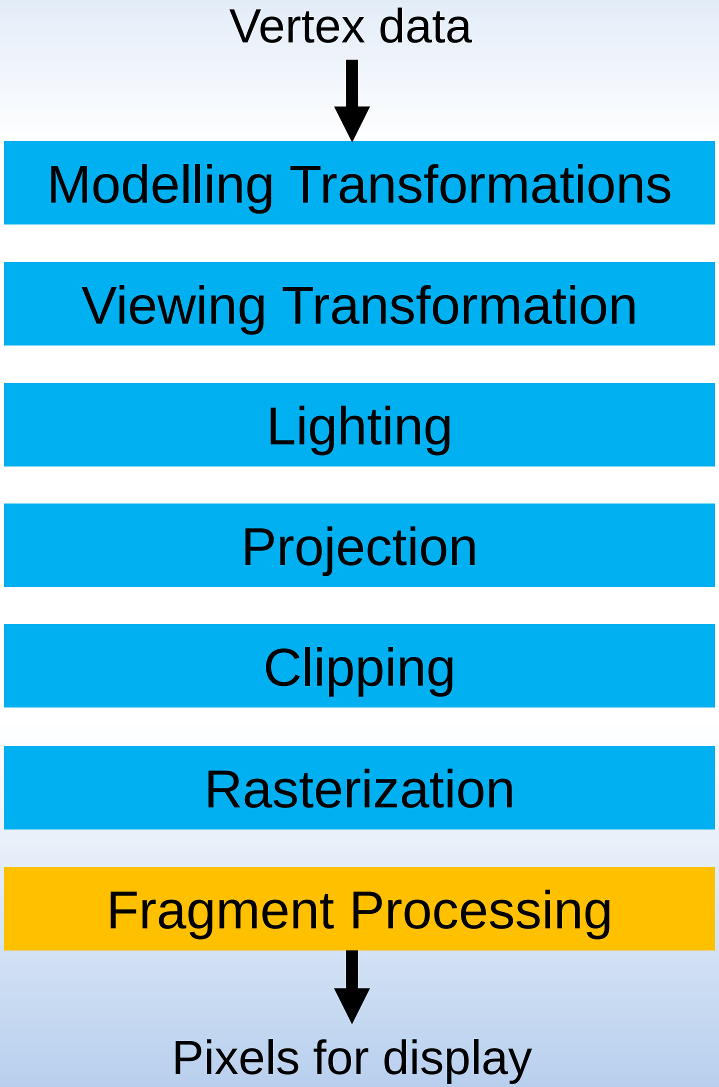
Discrete Techniques

The Computer Graphics Pipeline

- Fragment processing
  - ➢ Operations to determines final pixel colour
    - Buffer operations
      - Colour buffer, depth buffer, stencil buffer, etc.
    - Texture mapping
    - Blending
    - Per-pixel lighting



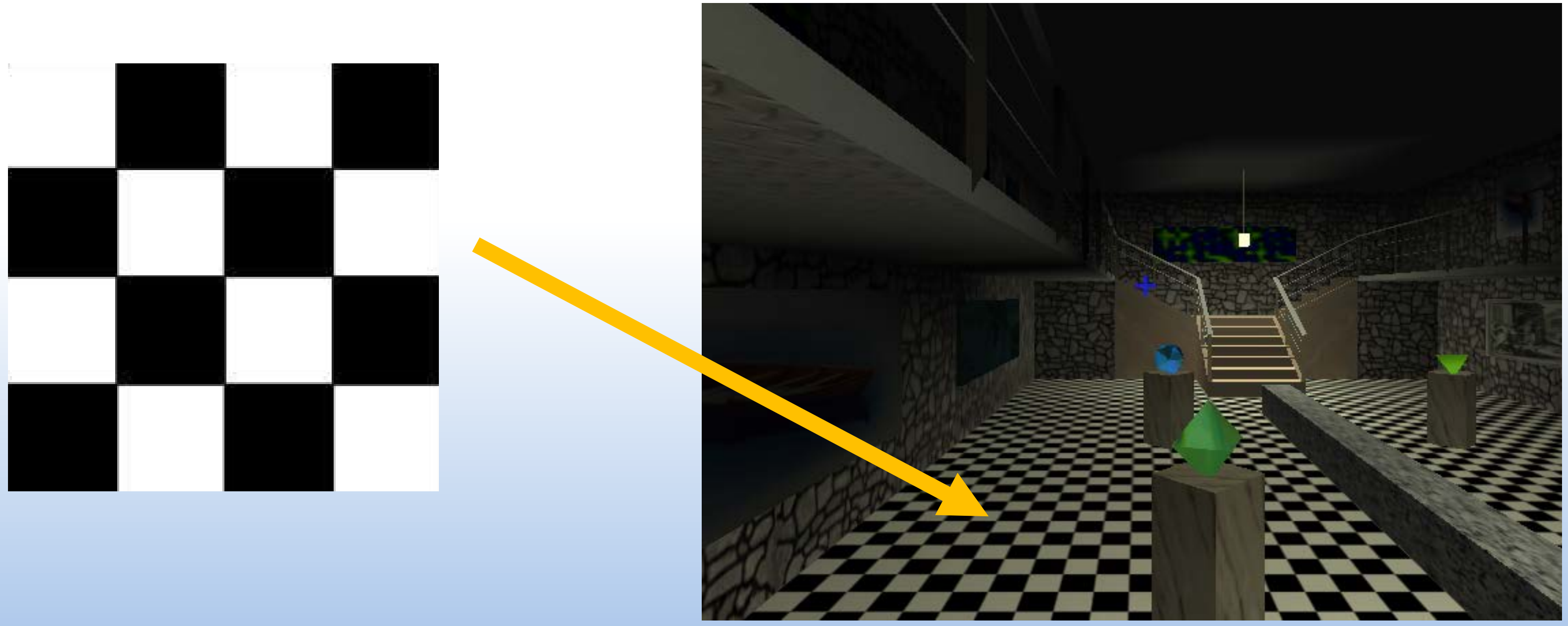


What are Fragments?

- After surface rasterized, no longer notion of a polygon
  - Surface essentially been “broken up” or “discretised” into small pieces
    - Each of which is at most the size of one pixel, called fragments
  - Picture elements, or pixels, are the final screen space colour of the rendered image
  - Conceptually fragments can be smaller than one pixel
    - More than one fragment can contribute to the colour of a pixel

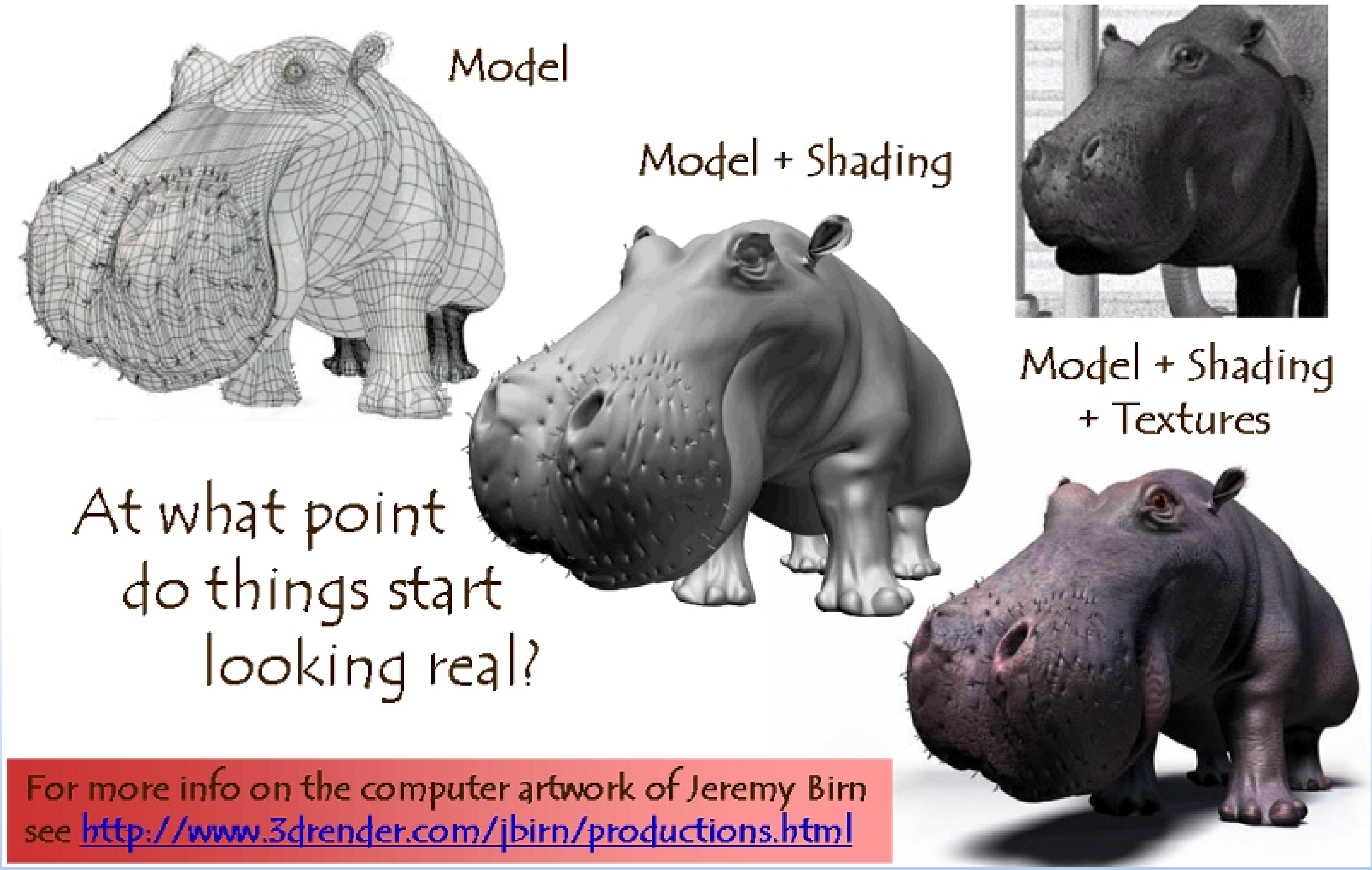
Mapping Techniques

- Texture mapping
  - ➢ There are limitation to geometric modeling
  - ➢ “Paint” image onto polygons
    - Analogy – sticking wallpaper onto a wall
  - ➢ Increases the visual realism



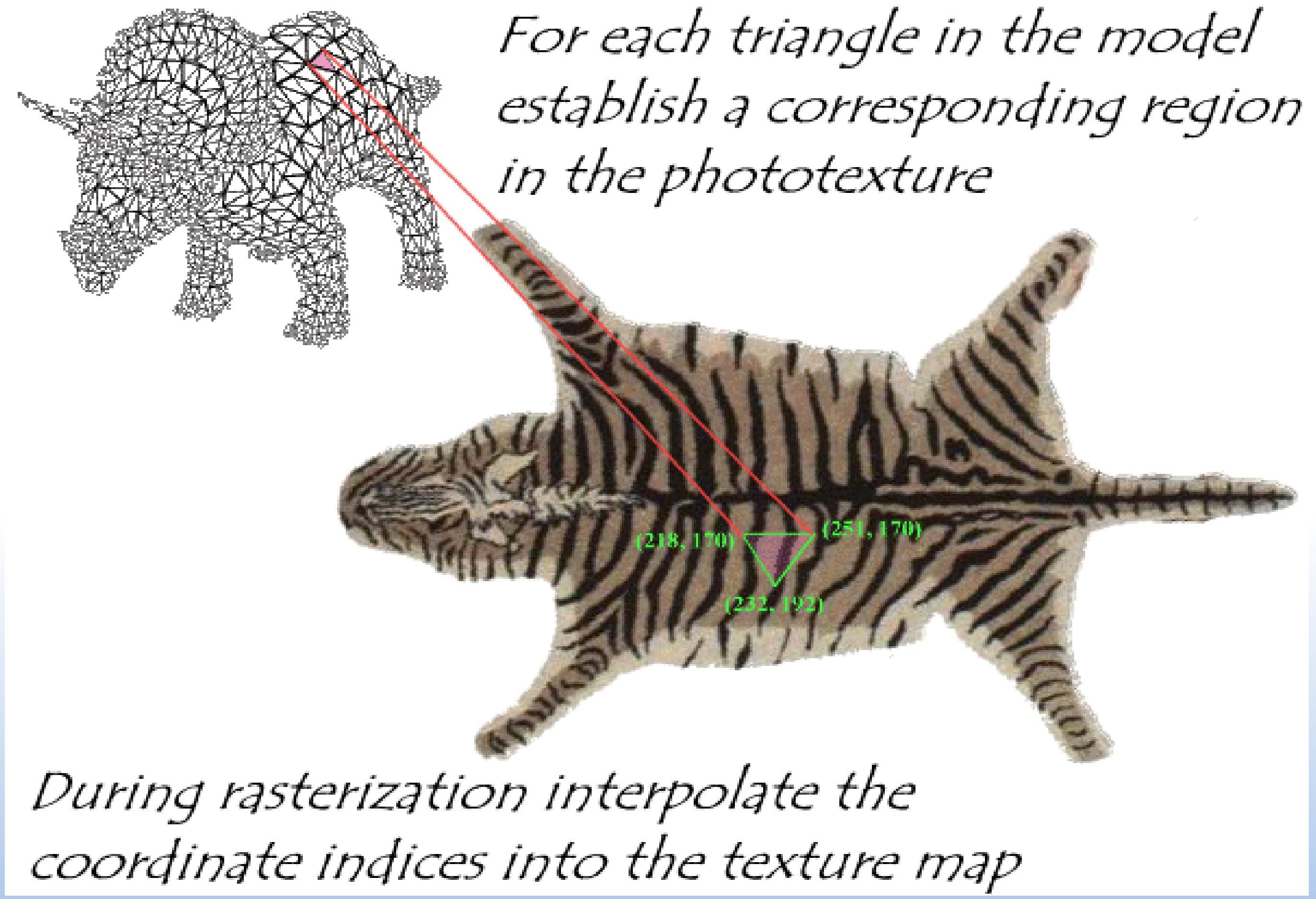
Mapping Techniques

- Texture mapping



Mapping Techniques

- Texture mapping

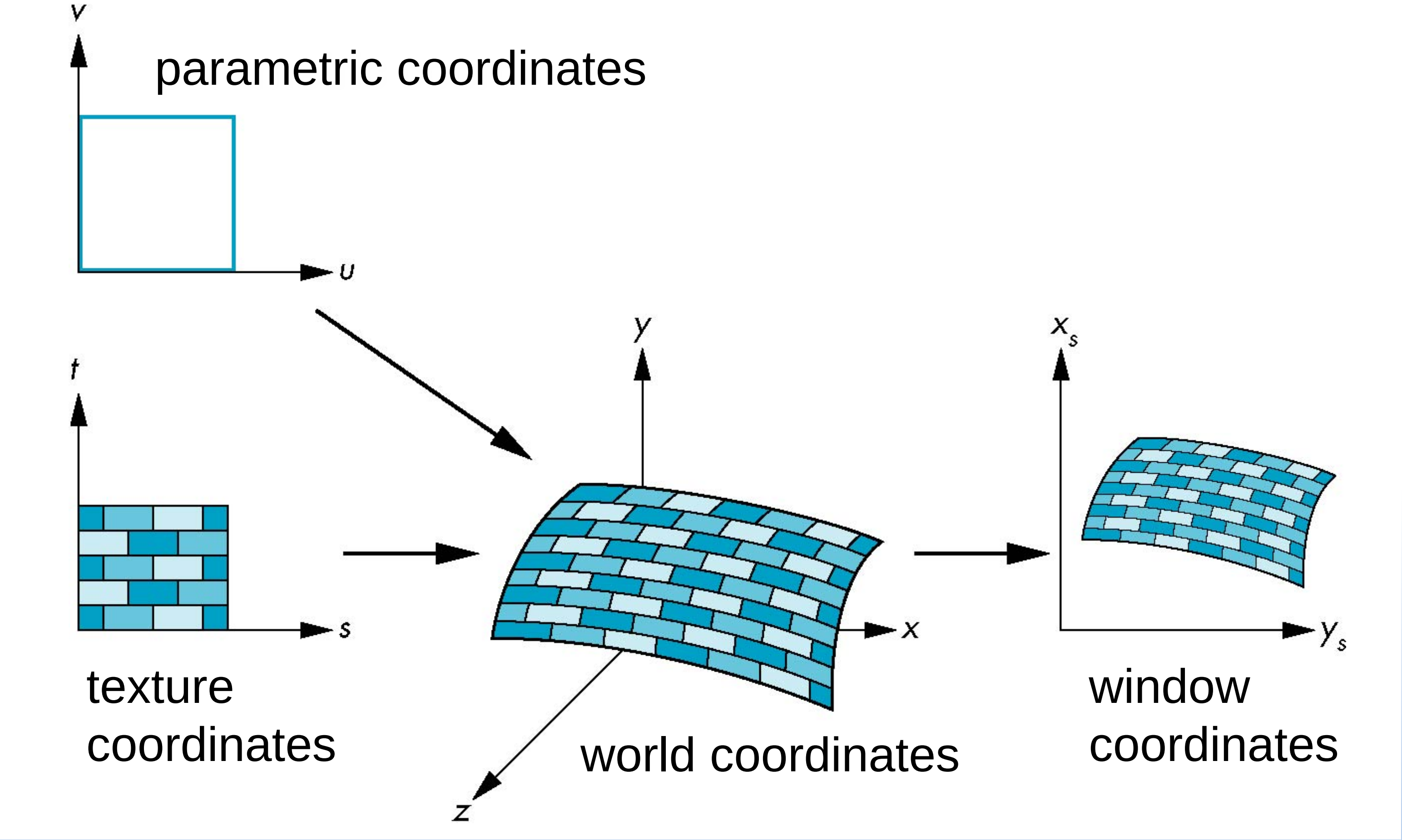


Mapping Techniques

- Texture mapping
  - Different texture maps can be used for same object

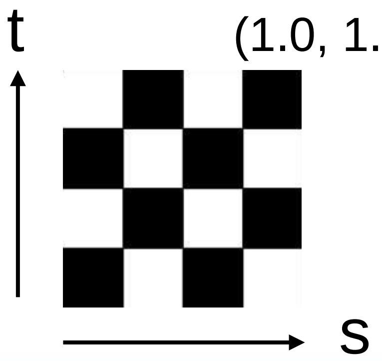
Mapping Techniques

- Texture mapping

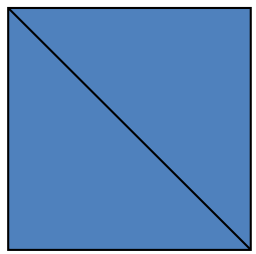


Mapping Techniques

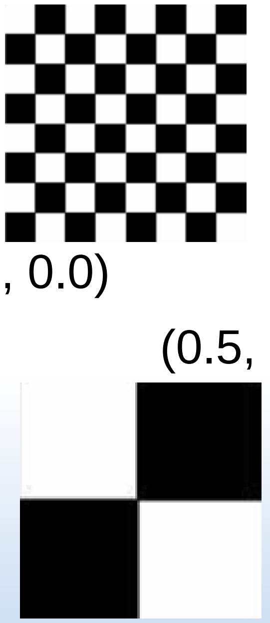
- Texture mapping
  - ➢ Texture coordinates are referred to as (s, t)



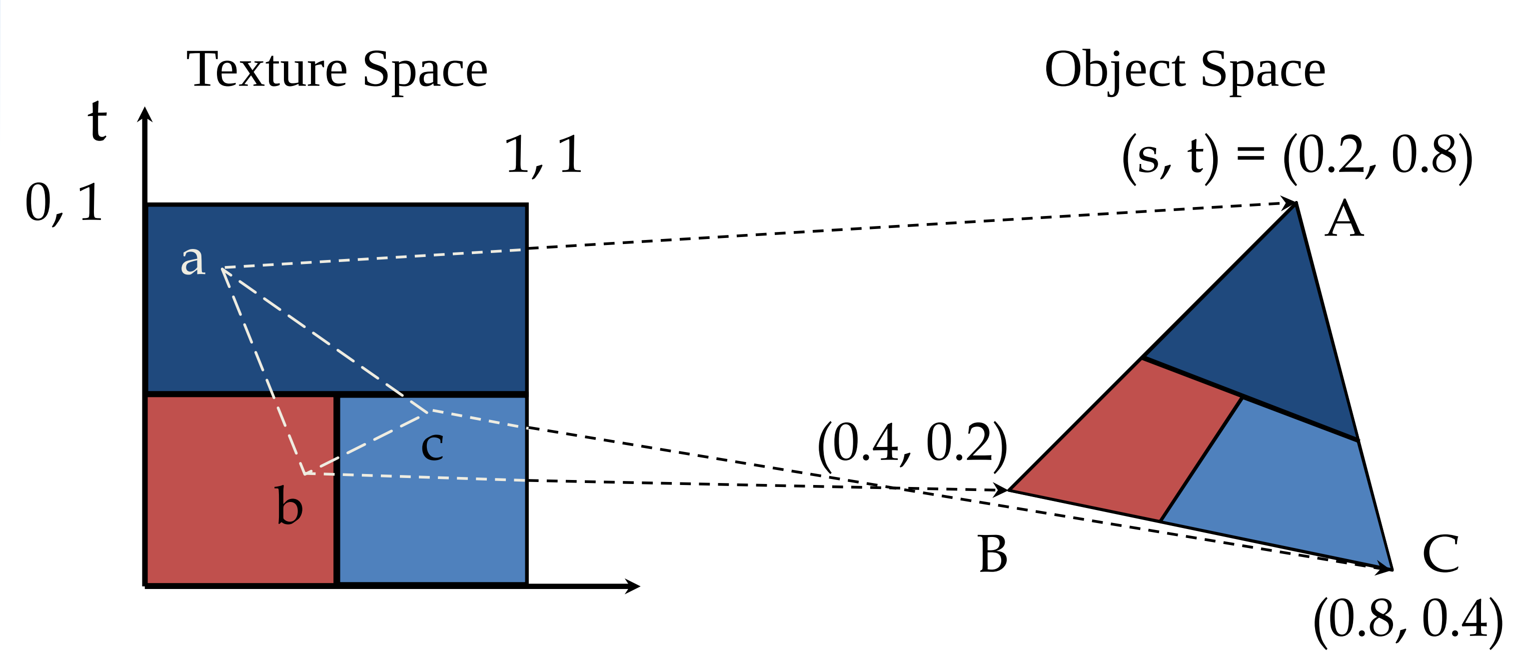
Texture

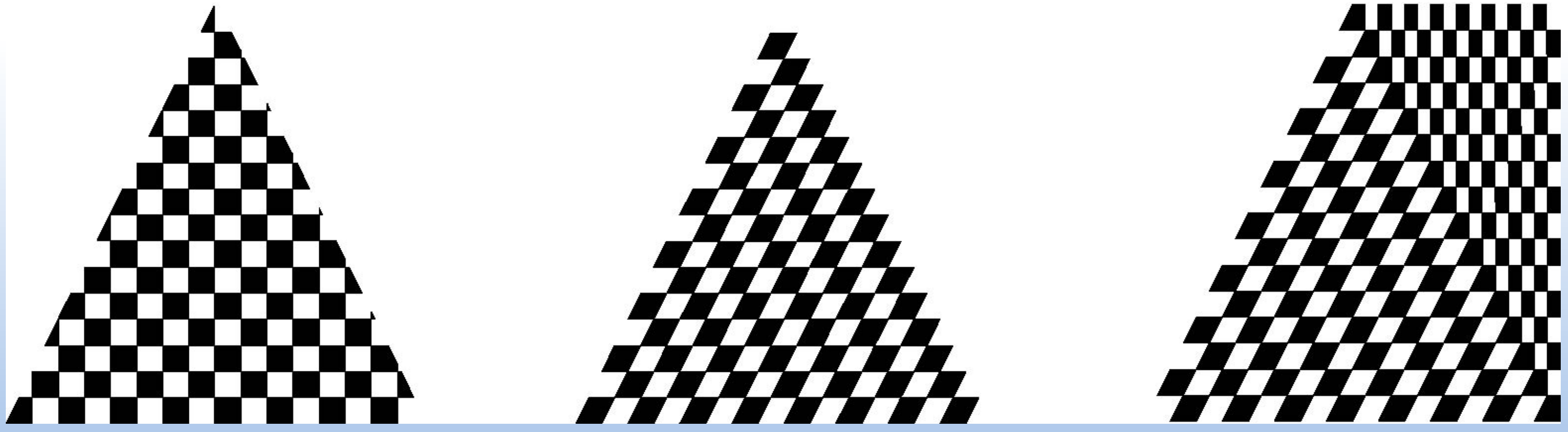


Polygon



Mapping Techniques





Mapping Techniques

- Texture coordinates

```
struct VertexTex {
   GLfloat position[3];
   GLfloat texCoord[2];
};

std::vector<GLfloat> gVertices = {
   -0.2f, -0.5f, 0.0f, // vertex 0: position
   0.0f, 0.0f,         // vertex 0: texture coordinate
   0.8f, -0.5f, 0.0f,  // vertex 1: position
   1.0f, 0.0f,         // vertex 1: texture coordinate
   -0.2f, 0.5f, 0.0f,  // vertex 2: position
   0.0f, 1.0f,         // vertex 2: texture coordinate
   0.8f, 0.5f, 0.0f,   // vertex 3: position
   1.0f, 1.0f,         // vertex 3: texture coordinate
};
```

Mapping Techniques

- Texture coordinates

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
  sizeof(VertexTex),
  reinterpret_cast<void*>(offsetof(VertexTex, position)));
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE,
  sizeof(VertexTex),
  reinterpret_cast<void*>(offsetof(VertexTex, texCoord)));


glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
```

Mapping Techniques

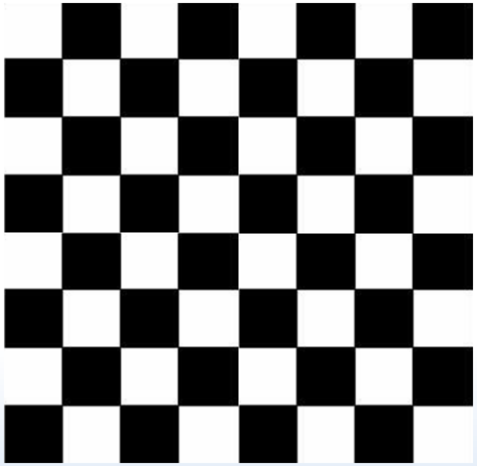
- Texture coordinates
  - In vertex shader

```
layout(location = 0) in vec3 aPosition;
layout(location = 1) in vec2 aTexCoord;

out vec2 vTexCoord;

void main()
{
...
   // interpolate texture coordinate
   vTexCoord = aTexCoord;
}
```

Mapping Techniques

- Texture mapping
  - ➢ Texture coordinates and wrapping

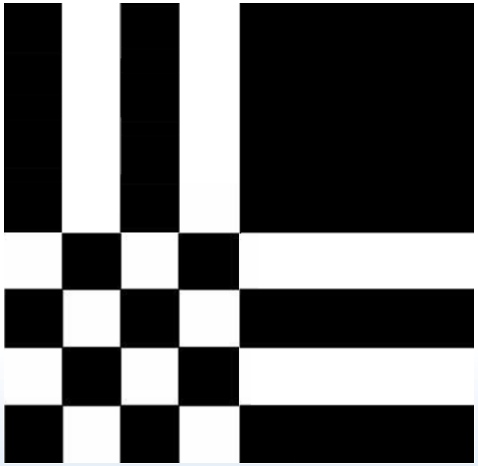
(2.0, 2.0)



(0.0, 0.0)

Repeat

(2.0, 2.0)



(0.0, 0.0)

Clamp

Mapping Techniques

- Texture mapping
  - ➢ Wrapping
    - Texture coordinates outside the 0.0 to 1.0 range
    - `GL_REPEAT` : Default, repeats the image
    - `GL_MIRRORED_REPEAT` : Mirrors each repeat
    - `GL_CLAMP_TO_EDGE` : Use edge colour
    - `GL_CLAMP_TO_BORDER` : Use a user defined border colour

GL_REPEAT GL_MIRRORED_REPEAT GL_CLAMP_TO_EDGE GL_CLAMP_TO_BORDER

https://learnopengl.com/Getting-started/Textures

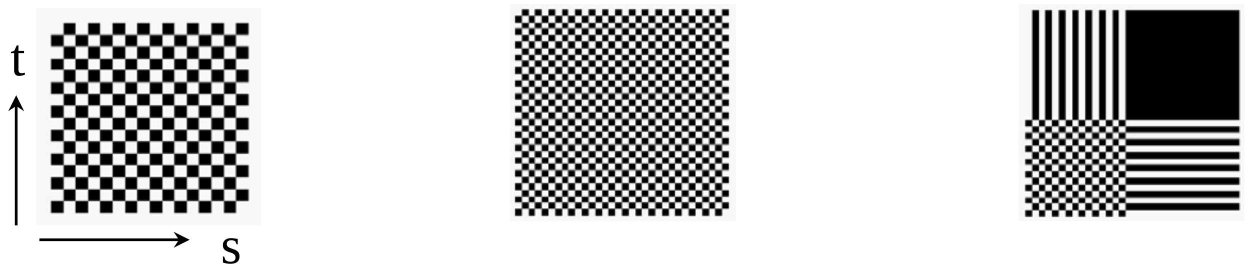
Mapping Techniques

➢ Wrapping

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
  GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
  GL_REPEAT);
```

- If clamping to border colour

```
float borderColor[] = { 1.0f, 0.0f, 0.0f, 1.0f };
glTexParameterfv(GL_TEXTURE_2D,
  GL_TEXTURE_BORDER_COLOR, borderColor);
```



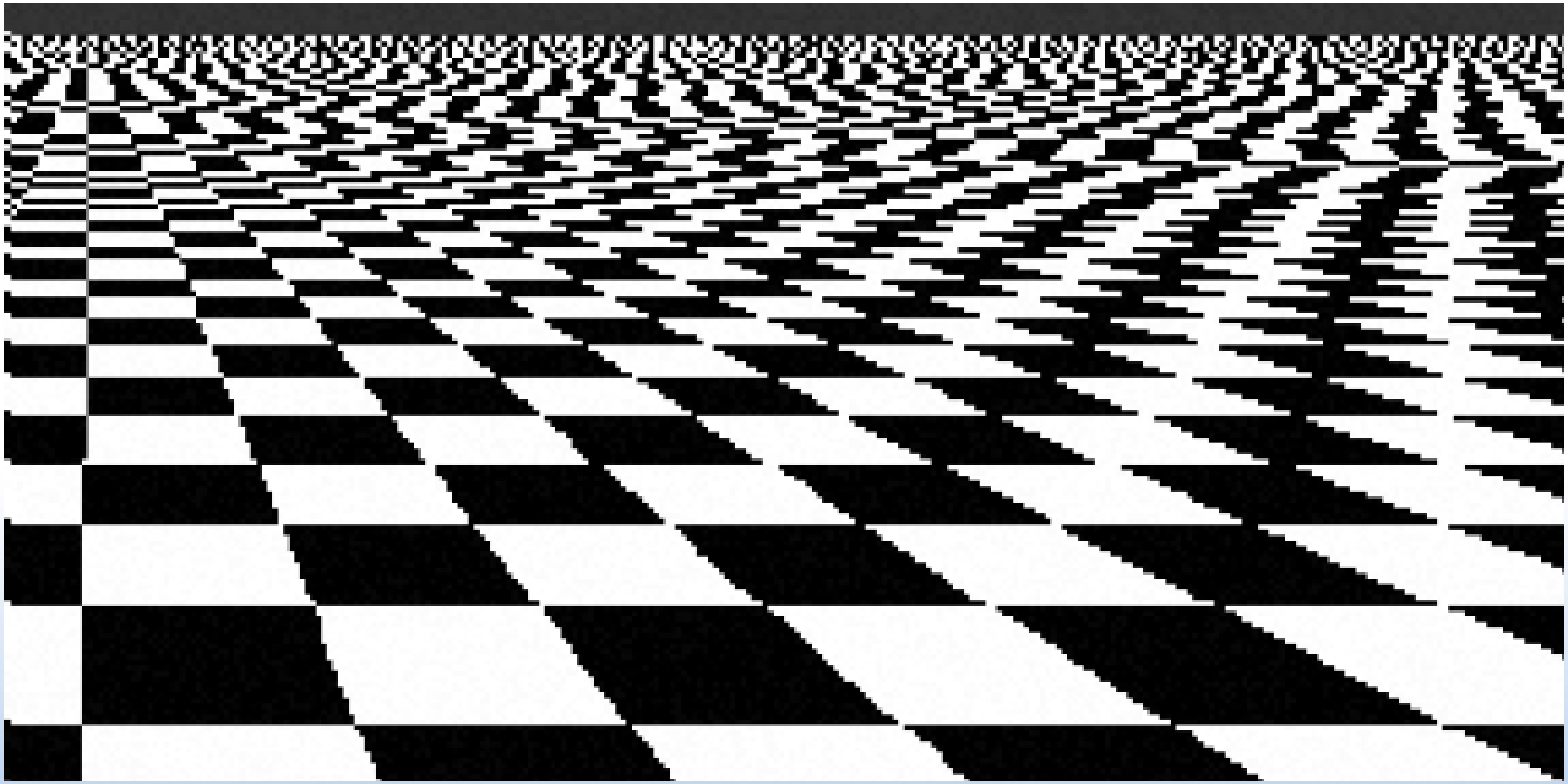
texture

GL_REPEAT wrapping

GL_CLAMP_TO_EDGE wrapping

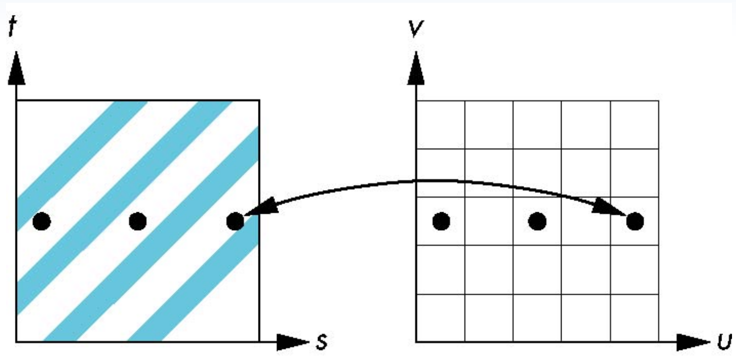
Mapping Techniques

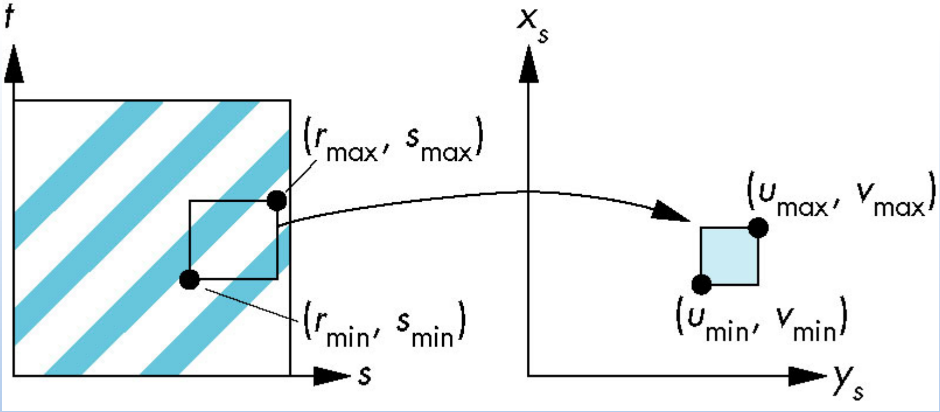
- Texture mapping
  - ➢ Aliasing in textures

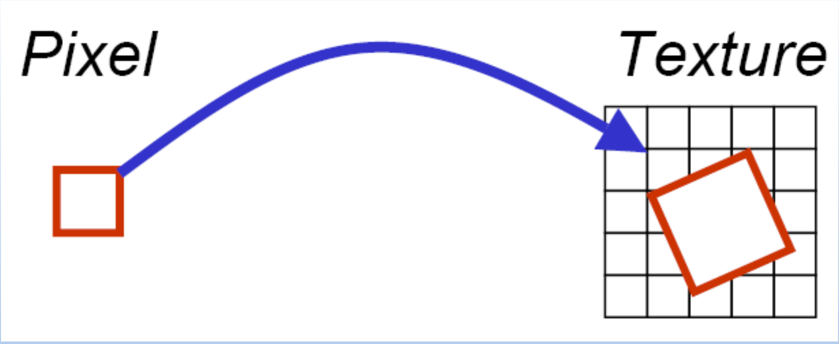


Mapping Techniques

- Texture mapping
  - Aliasing is the under-sampling of a signal
  - In texture mapping, aliasing artifacts emerge as a result of discrete sampling
  - Looks worse when moving





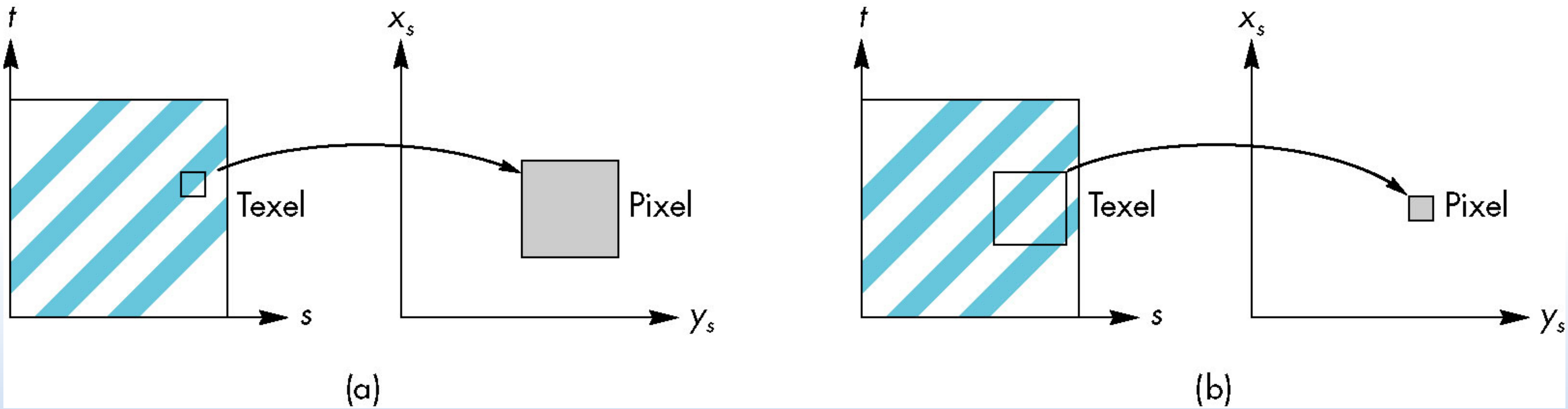


Mapping Techniques

- Texture mapping
  - Size of pixel we are trying to colour may be larger or smaller than one texture element (**texel**)

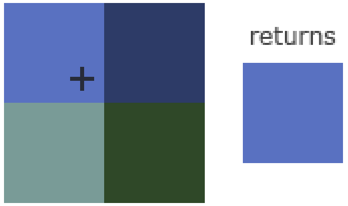
Magnification

Minification

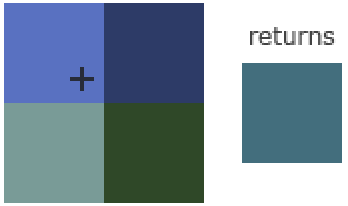


Mapping Techniques

- Texture mapping
  - ➢ Filtering
    - GL_NEAREST : Default filtering method
      - – Nearest neighbour filtering
        - » Select colour closest to texture coordinate



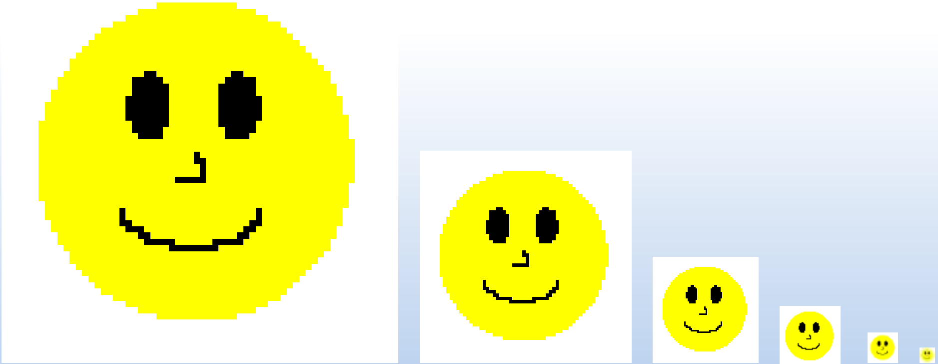
    - GL_LINEAR
      - – (bi)linear filtering
        - » Interpolate value from neighbouring texels
        - » The closer to a colour, the greater its contribution

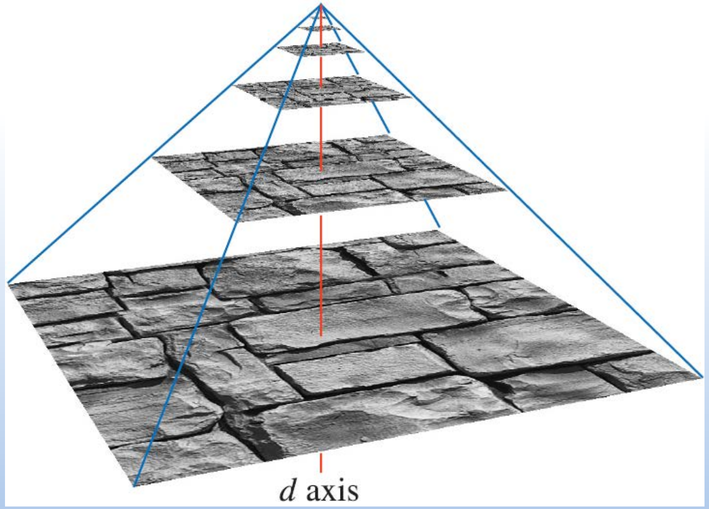


https://learnopengl.com/Getting-started/Textures

Mapping Techniques

- Texture mapping
  - Mipmaps
    - For minification problem
    - Create multiple resolutions of an image
      - E.g., for a 256 x 256 image
        - » Create 128x128, 64x64, 32x32, 16x16, 8x8, 4x4, 2x2, 1x1
      - Sample used depends on polygon's size on screen



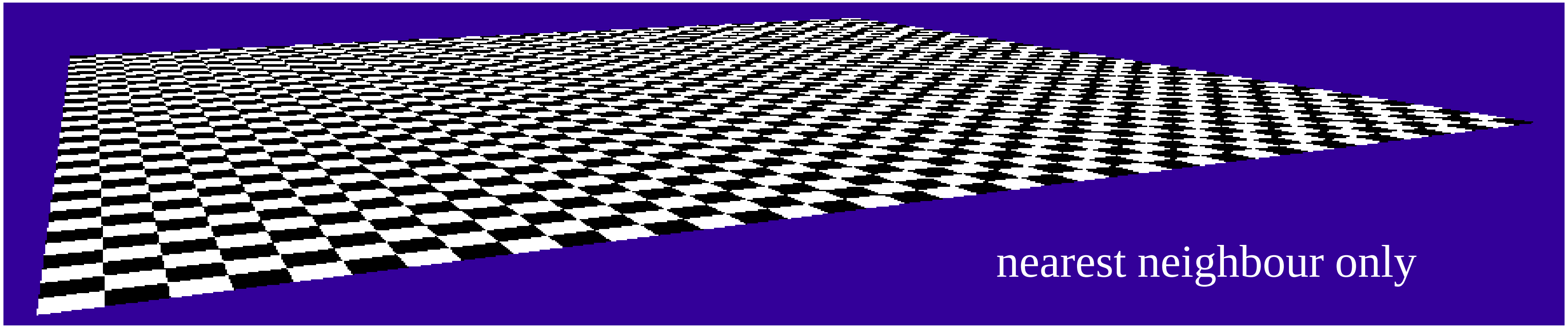


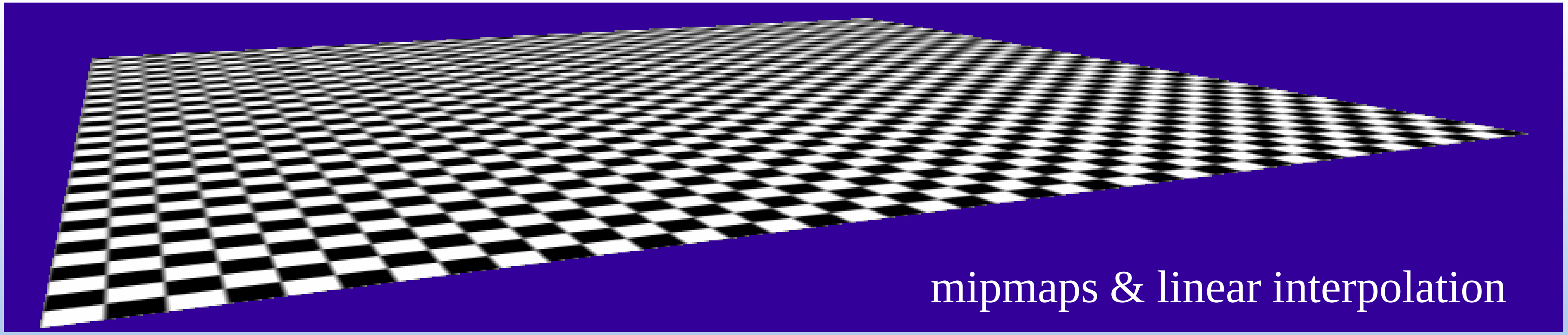
Mapping Techniques

- Mipmap filtering
  - Artifacts like sharp edges may be visible when switching between mipmap layers
  - Filter between mipmap levels
    - `GL_NEAREST_MIPMAP_NEAREST` : Nearest mipmap to match the pixel size and nearest neighbor interpolation for texture sampling
    - `GL_LINEAR_MIPMAP_NEAREST` : Nearest mipmap level and samples that level using linear interpolation
    - `GL_NEAREST_MIPMAP_LINEAR` : Linearly interpolates between the two mipmaps that most closely match the size of a pixel and samples the interpolated level via nearest neighbor interpolation
    - `GL_LINEAR_MIPMAP_LINEAR` : Linearly interpolates between the two closest mipmaps and samples the interpolated level via linear interpolation

Mapping Techniques

- Texture mapping
  - Mipmaps can be used in conjunction with texture filtering methods





Mapping Techniques

- Generating a texture
  - Using an image library

```
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
```

  - Load image data

```
stbi_set_flip_vertically_on_load(true);

int imageWidth, imageHeight, imageChannels;
unsigned char* imageData =
    stbi_load("./images/check.bmp",
    &imageWidth, &imageHeight, &imageChannels, 0);
```

Mapping Techniques

- Generating a texture
  - ➢ Generate OpenGL texture object

```
GLuint gTextureID;
…
glGenTextures(1, &gTextureID);
glBindTexture(GL_TEXTURE_2D, gTextureID);
```

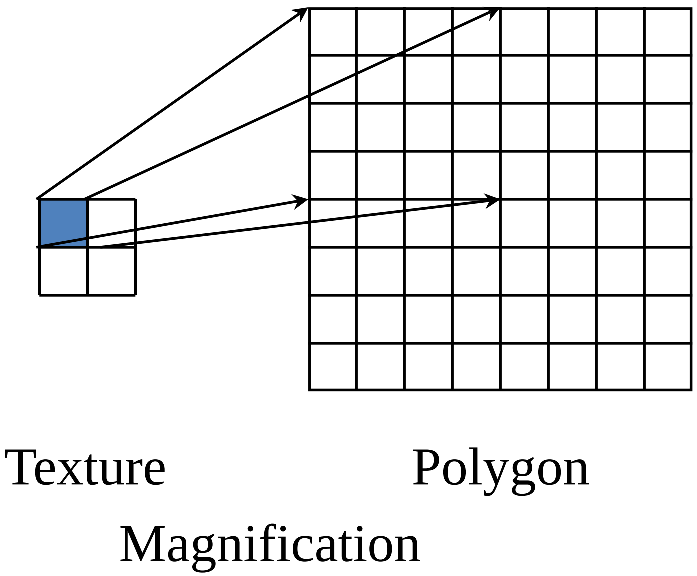
  - ➢ Create texture and mipmaps

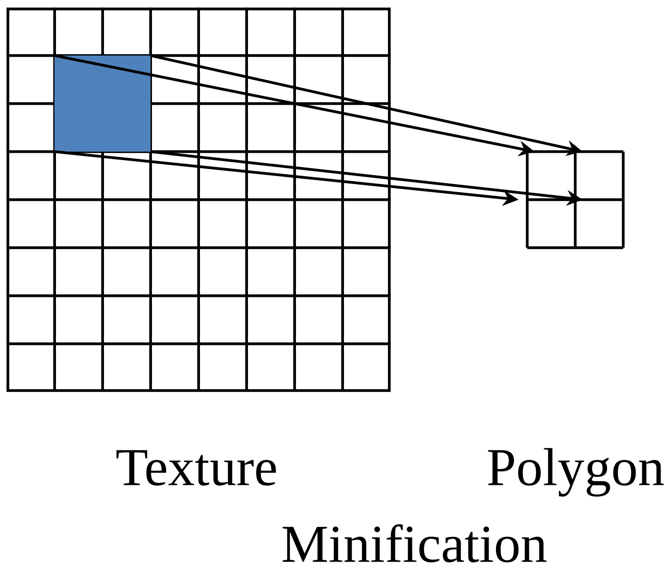
```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, imageWidth,
    imageHeight, 0, GL_RGB, GL_UNSIGNED_BYTE,
    imageData);
glGenerateMipmap(GL_TEXTURE_2D);
```

Mapping Techniques

- Generating a texture
  - Set texture parameters

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXURE_MAX_FILTER,
   GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXURE_MIN_FILTER,
   GL_NEAREST_MIPMAP_NEAREST);
```





Mapping Techniques

- Using the texture
  - In fragment shader

```
in vec2 vTexCoord;

uniform sampler2D uTextureSampler;

out vec3 fColor;

void main()
{
    fColor = texture(uTextureSampler, vTexCoord).rgb;
}
```

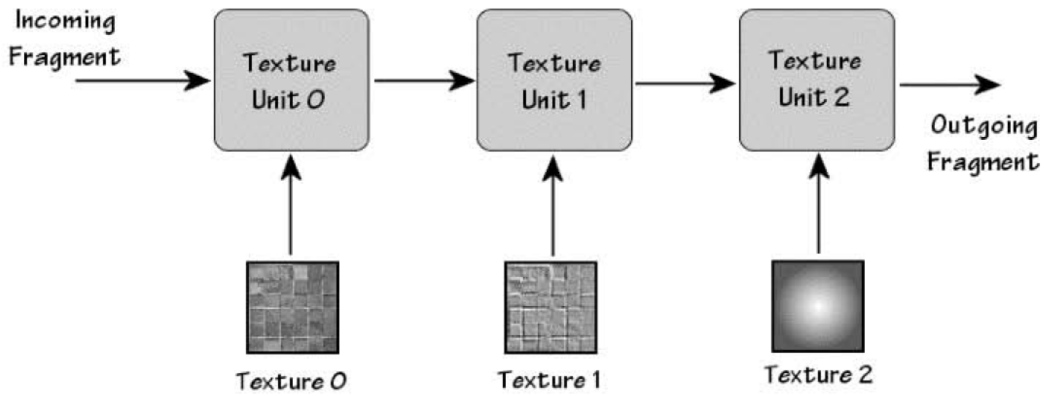
Mapping Techniques

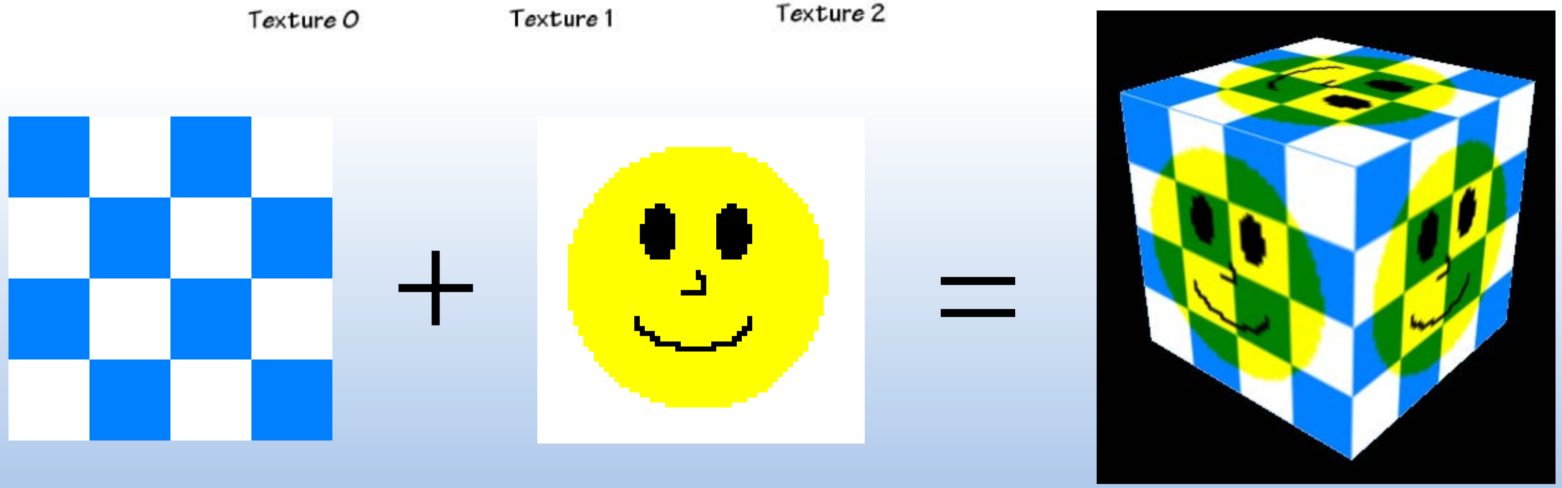
- Using the texture
  - In the render_scene() function

```
gShader.use();

// set texture
gShader.setUniform("uTextureSampler", 0);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, gTextureID);

glBindVertexArray(gVAO);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
```

Mapping Techniques

- Multi-texturing
  - Paint multiple textures onto the same surface



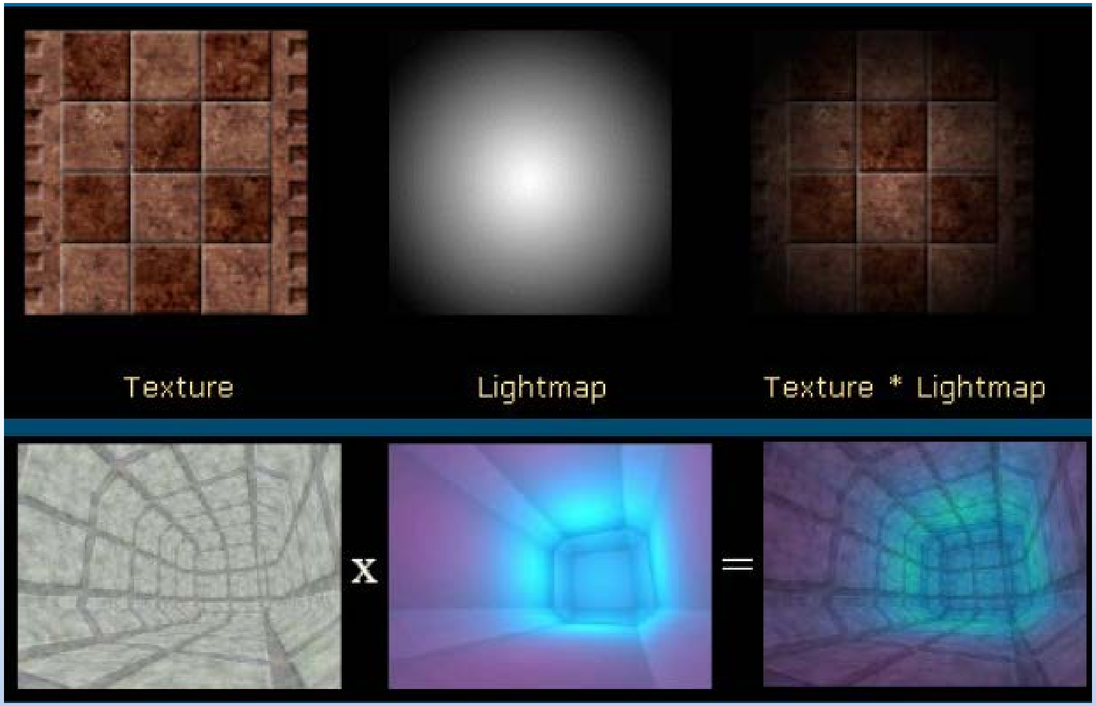


Mapping Techniques

- Light mapping
  - Illumination maps/light maps were introduced in the game *Quake* to represent lighting effects

Mapping Techniques

- Light mapping
  - Allows lighting to be pre-calculated and stored as 2D texture map



Quake 2's packed light map texture

Mapping Techniques

- Light mapping

X

=

Quake 2 light mapping

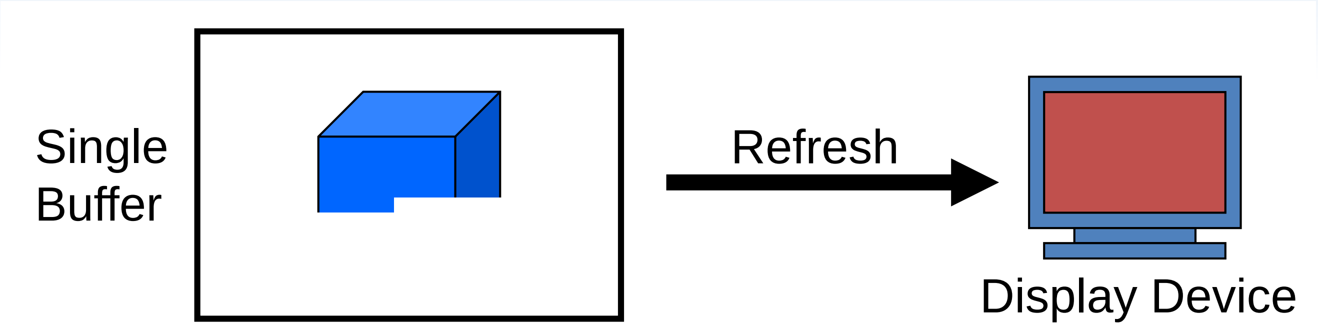
Buffers

- Per-pixel data is stored in buffers
  - Colour Buffer (front and back)
  - Depth Buffer
  - Stencil Buffer
  - Others (overlay planes, auxiliary buffers, colour indices)

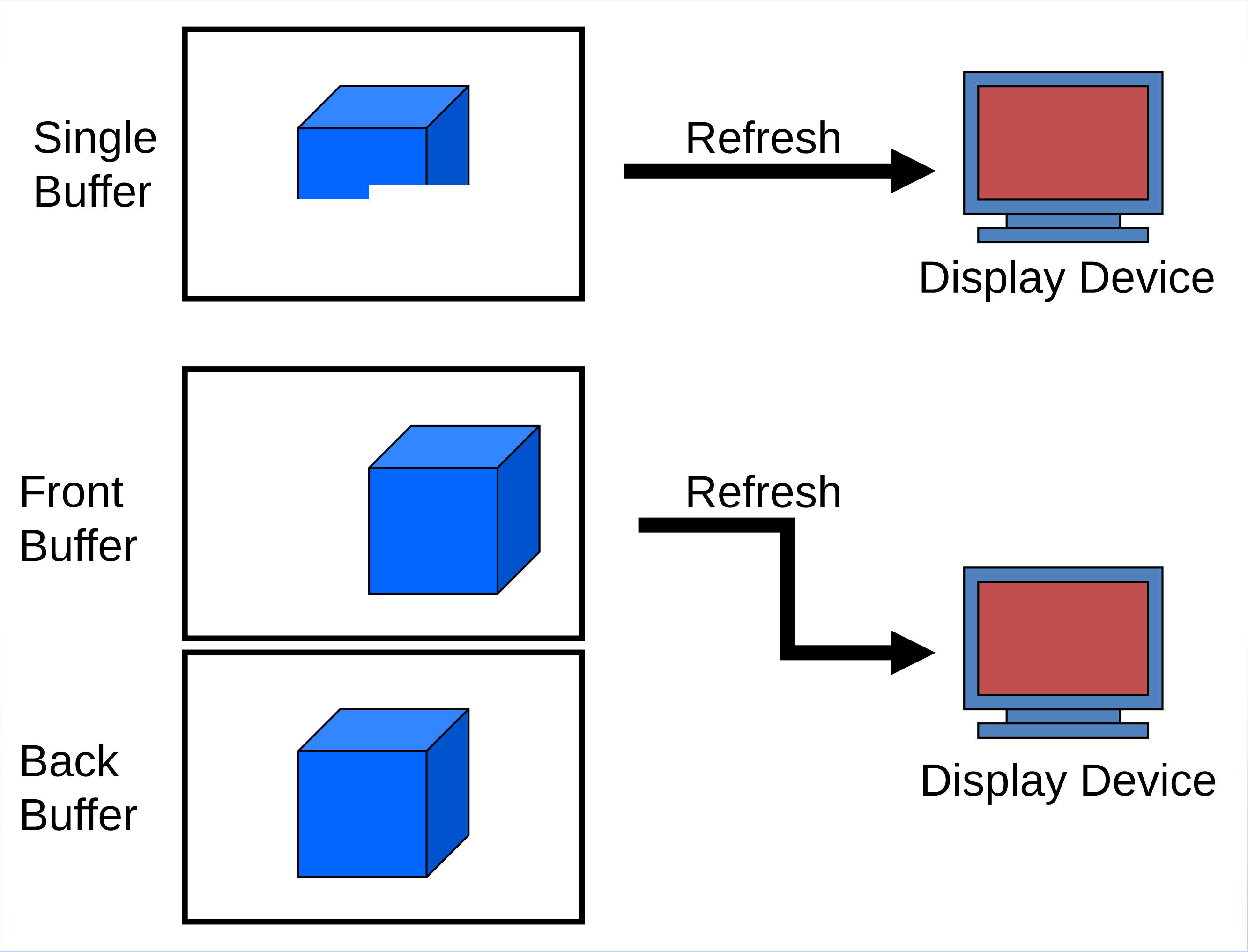
Buffers

- The colour buffer
  - Where the RGBA pixel values are stored
  - Generally need two separate colour buffers
    - One for displaying, the other for rendering
    - In double buffering referred to as front and back buffers
  - Double buffering
    - To prevent flickering and other undesirable artifacts that will appear if an image that is currently being displayed is updated

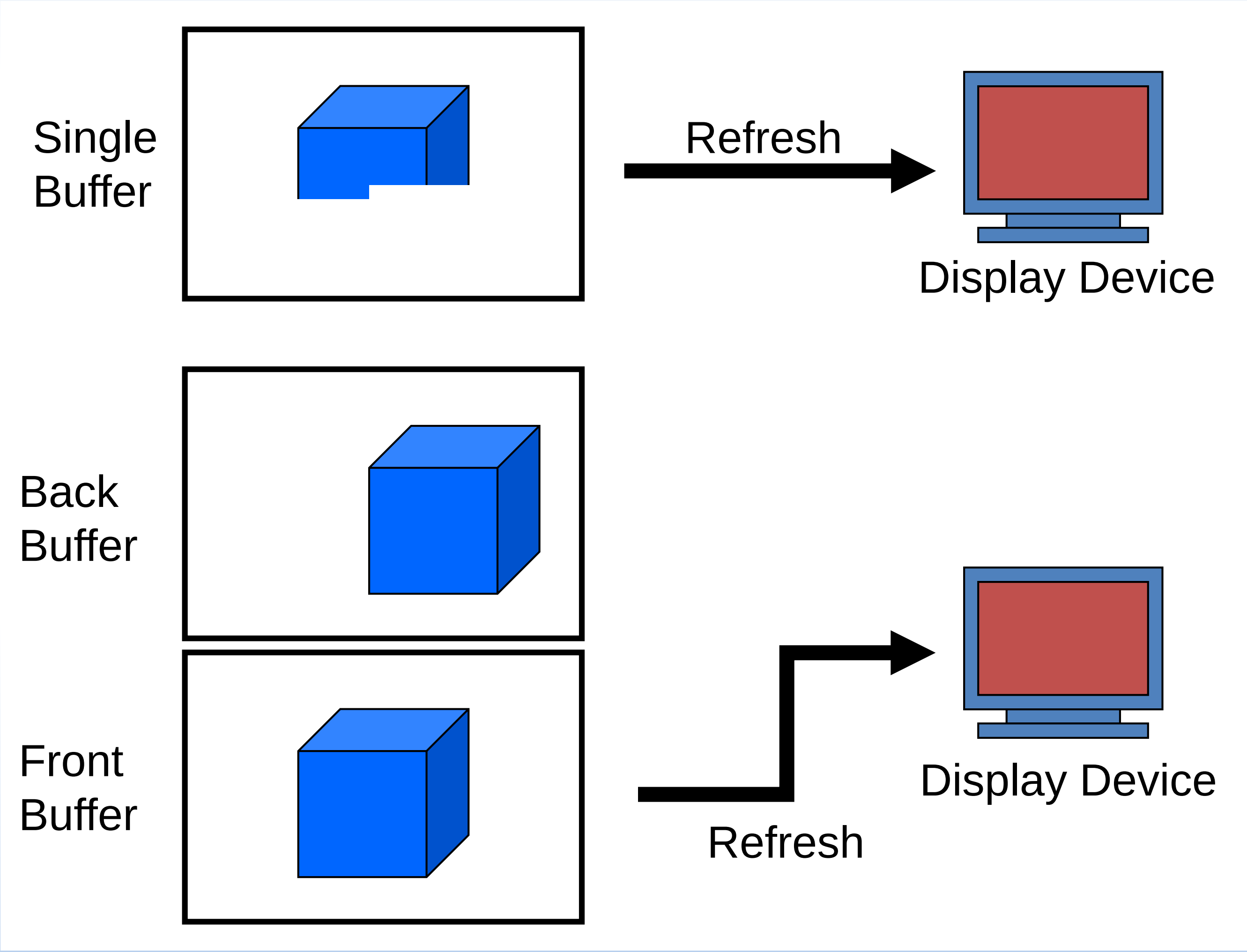
Single Buffer



Double Buffering

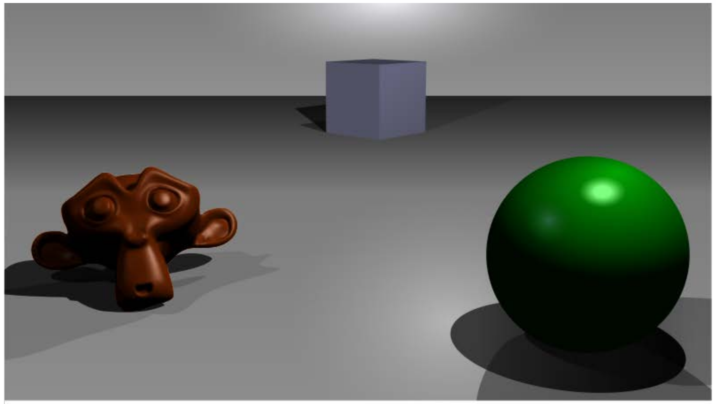


Double Buffering

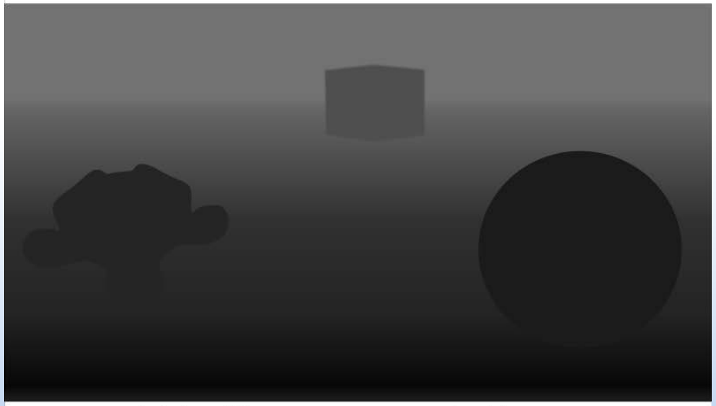


Buffers

- The depth-buffer (z-buffer)
  - ➢ Use for visibility determination
  - ➢ Tests can be enabled or disabled
    - Certain situations need to disable depth-buffer tests
      - – e.g., when rendering translucent polygons



A simple three-dimensional scene



Z-buffer representation

Buffers

- The stencil buffer
  - General purpose buffer for doing things not possible with colour and depth buffer alone
    - E.g., for creating reflective surfaces and essential in some shadow rendering techniques, like shadow volumes
  - “Tag” pixels in one rendering pass to control their update in subsequent rendering passes
  - Can specify different rendering operations like
    - Stencil test fails
    - Stencil test passes & depth test fails
    - Stencil test passes & depth test passes

Buffers

- Clearing buffers

```
// set values to clear to
glClearColor(r, g, b, a);
glClearDepth(depth);
glClearStencil(value);

// clearing
// for example, at the start of render_scene()
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
  GL_STENCIL_BUFFER_BIT);
```

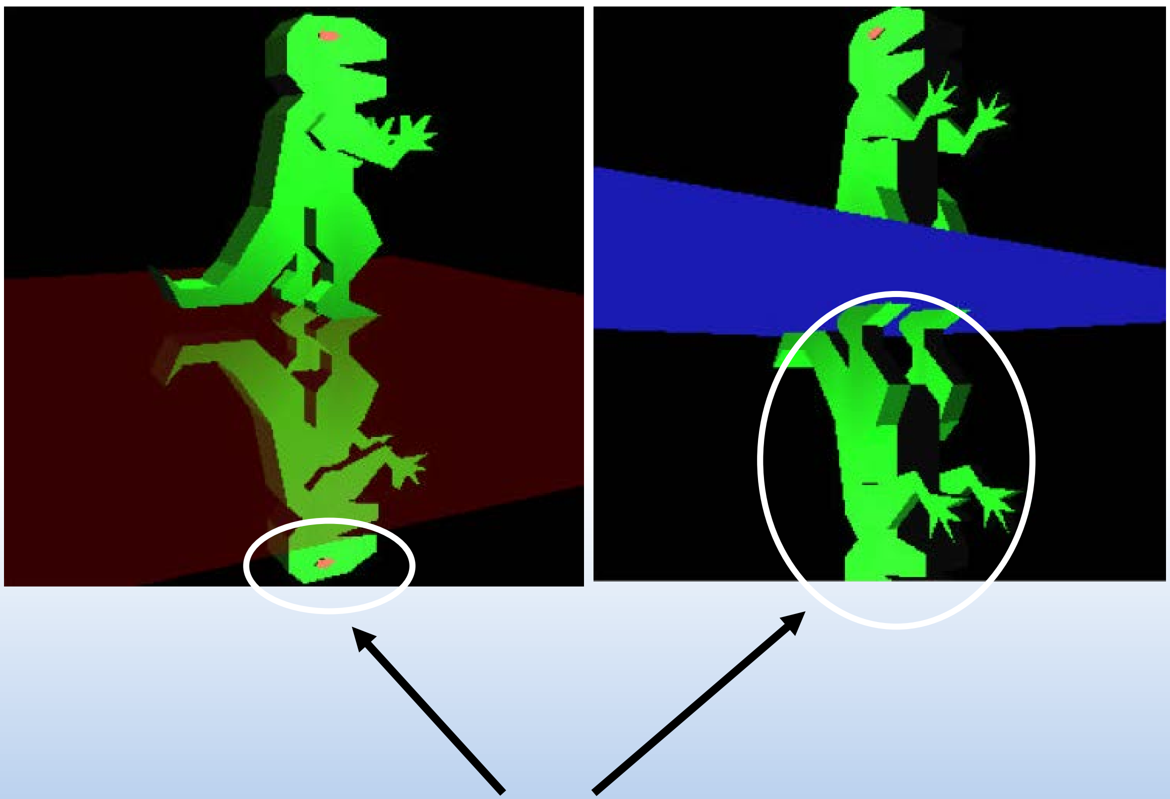
Buffers

- Multipass rendering
  - Render the scene multiple times
  - Composite the images in certain ways to achieve the desired effects in final scene
  - Two things to consider for each stage of a multipass algorithm
    - How to render the scene
    - How to composite the images

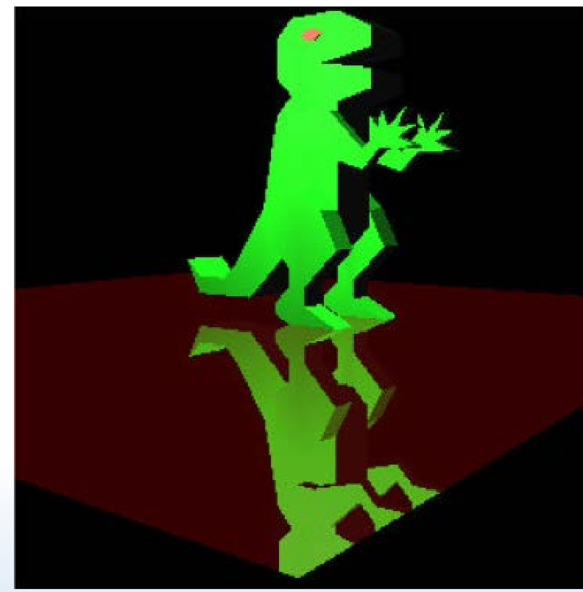
Buffers

- Stencil buffer for reflection

Without the stencil buffer

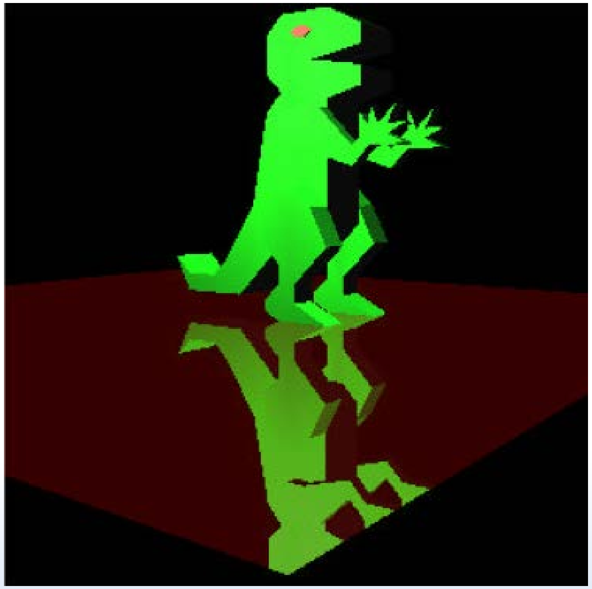


With the stencil buffer



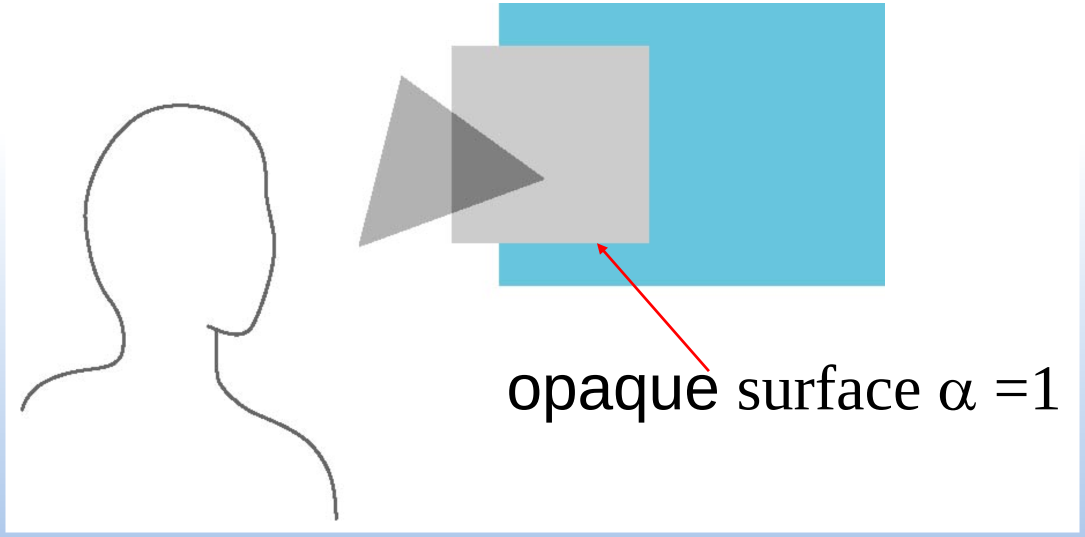
Buffers

- Stencil buffer for reflection
  - Basic algorithm
    - Clear buffers
    - Disable depth buffer and draw mirror surface to stencil buffer
    - Enable depth buffer, draw reflected geometry where stencil buffer passes
    - Disable stencil buffer, draw normal scene



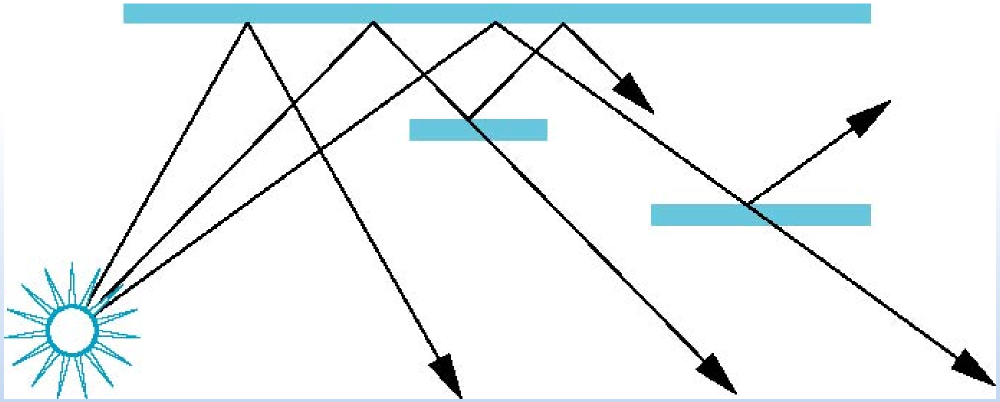
Compositing and Blending

- Alpha blending
  - ➢ Alpha channel
    - The 'A' component in the RGBA (or RGBα) colour mode
  - ➢ α values range from 0.0 to 1.0
    - Value of 1.0, surface completely opaque and blocks all incident light
    - Translucency of a surface given by 1.0 – α



Compositing and Blending

- Dealing with translucency in a physically correct manner is difficult due to
  - The complexity of the internal interactions of light and matter
  - Using a pipeline renderer



Compositing and Blending

- Blending equation
  - Can define source and destination blending factors for each RGBA component

    $\mathbf{s} = [s_r, s_g, s_b, s_a]$

    $\mathbf{d} = [d_r, d_g, d_b, d_a]$

  Suppose that the source and destination colors are

    $\mathbf{a} = [a_r, a_g, a_b, a_a]$

    $\mathbf{b} = [b_r, b_g, b_b, b_a]$

  Blend as

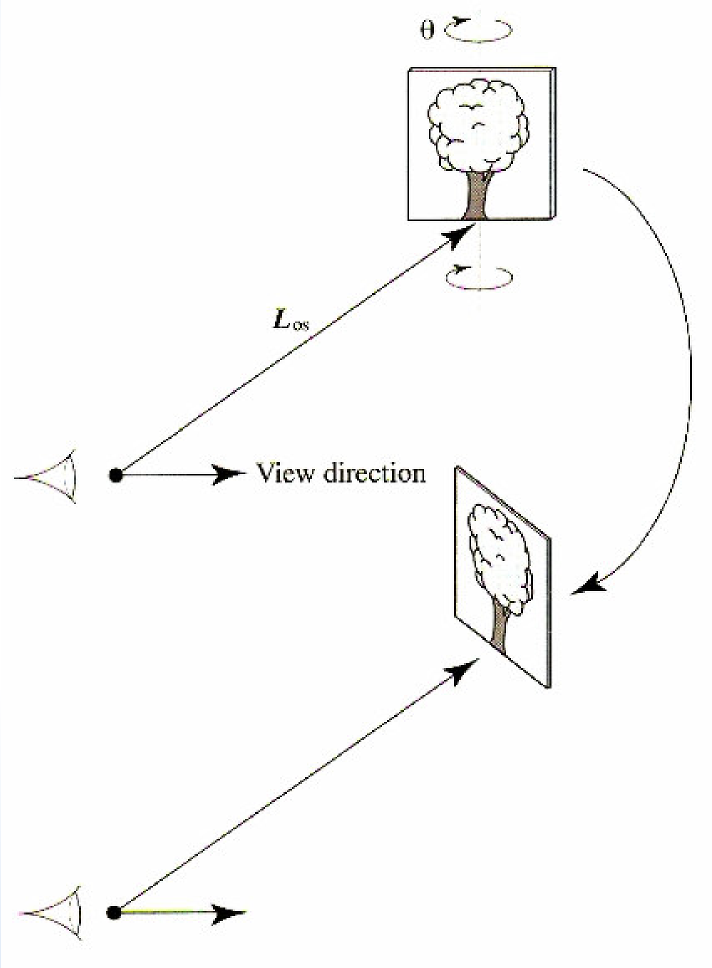
    $\mathbf{c}' = [a_r\, s_r + b_r\, d_r,\ a_g\, s_g + b_g\, d_g\ ,\ a_b\, s_b + b_b\, d_b\ ,\ a_a\, s_a + b_a\, d_a\ ]$

Compositing and Blending

- Hidden surface removal
  - Opaque polygons block all polygons behind them and affect the depth buffer
  - Translucent polygons should not affect depth buffer
  - Sort polygons first to remove order dependency
  - Easiest solution (not necessarily the best)
    - Render all opaque polygons first
    - Disable the depth test (or make the depth-buffer read-only)
    - Render translucent polygons in a back-to-front order

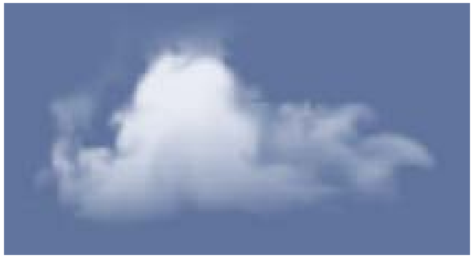
Mapping Techniques

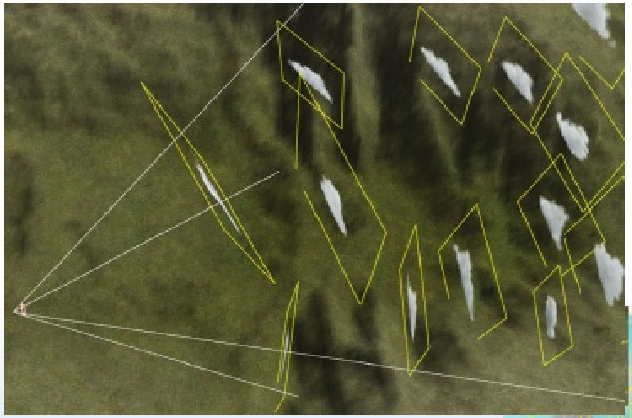
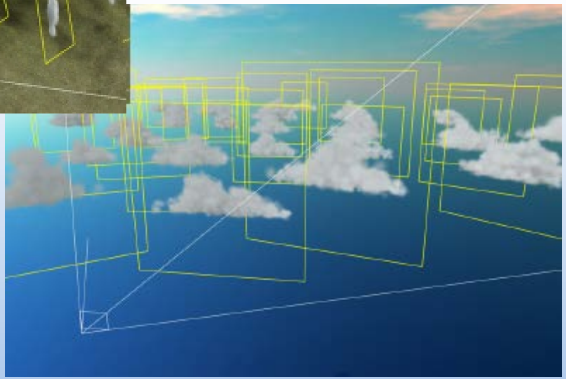
- Billboarding
  - 2D texture map used as a 3D entity in the scene
  - Rotated so polygon's normal always faces the viewer
  - Creates illusion 2D image is 3D object



Mapping Techniques

- Billboarding



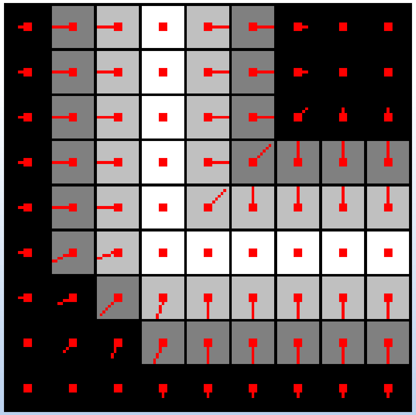



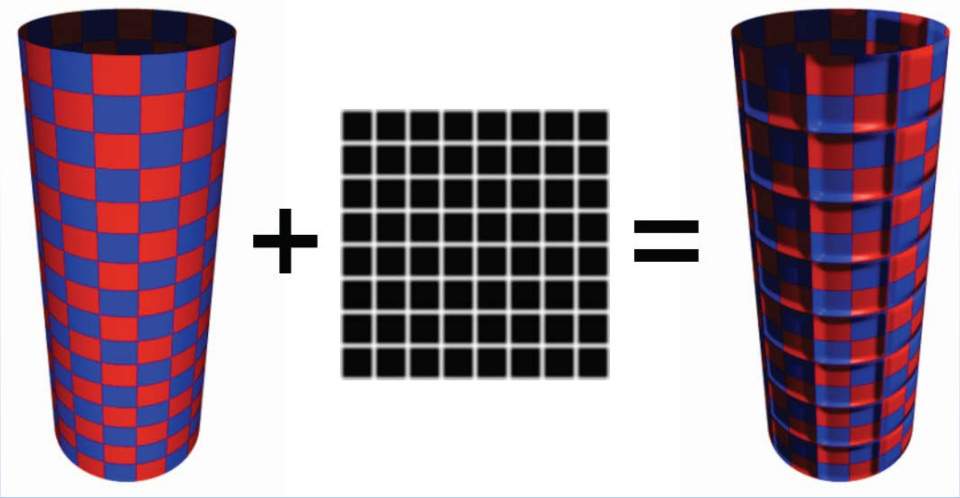

Mapping Techniques

- Normal mapping
  - Textures used to perturb the surface normal
    - Actual geometry of surface does not change, just shaded as if it were different shape

Mapping Techniques

- Normal mapping
  - Outline looks smooth

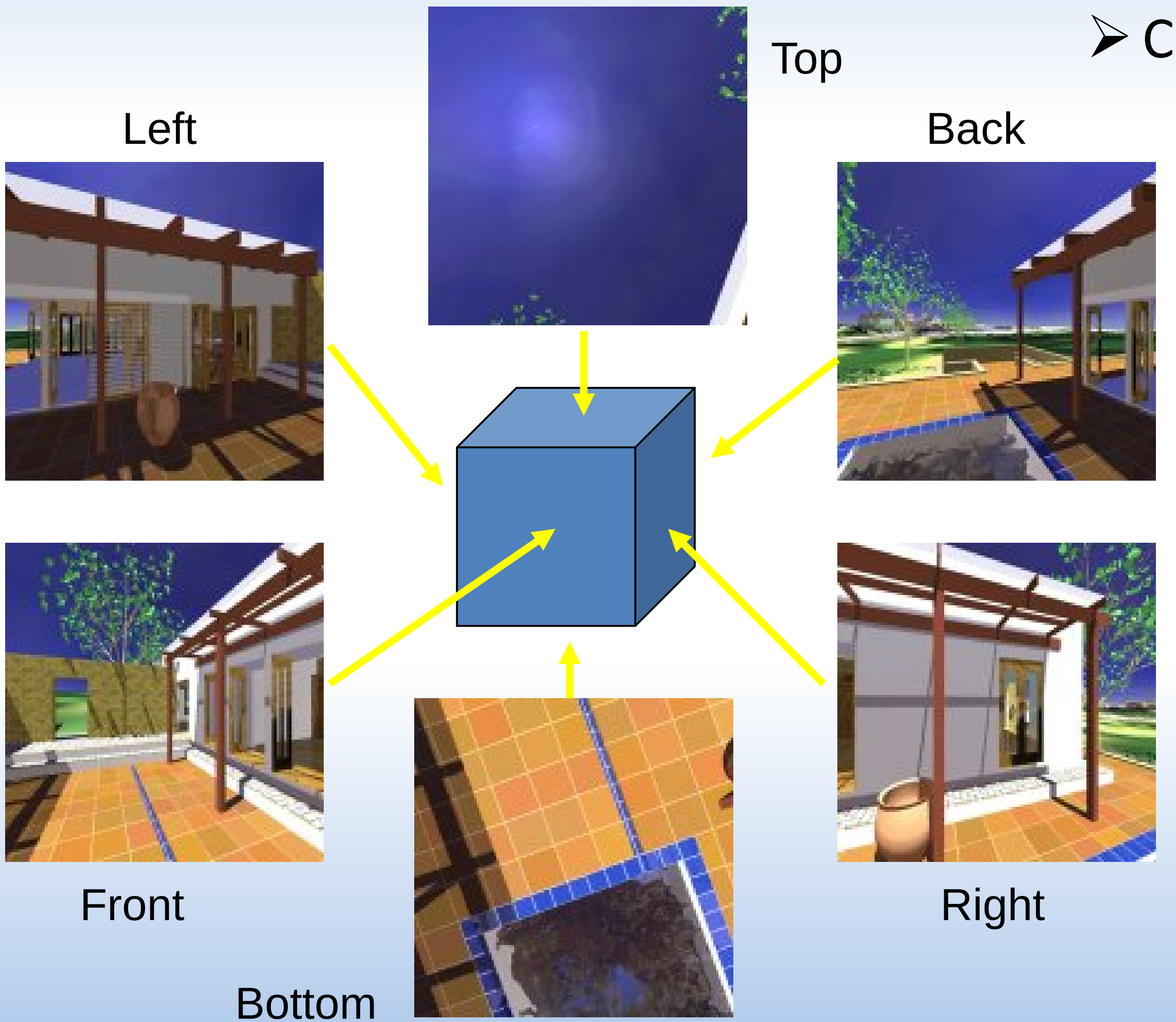




Mapping Techniques

- Environment mapping
  - Quick but inaccurate way of generating effects like reflections (a.k.a reflection mapping)
  - Image of the surrounding environment used to project reflections onto object's surface
  - Common methods
    - Cube mapping
    - Spherical mapping

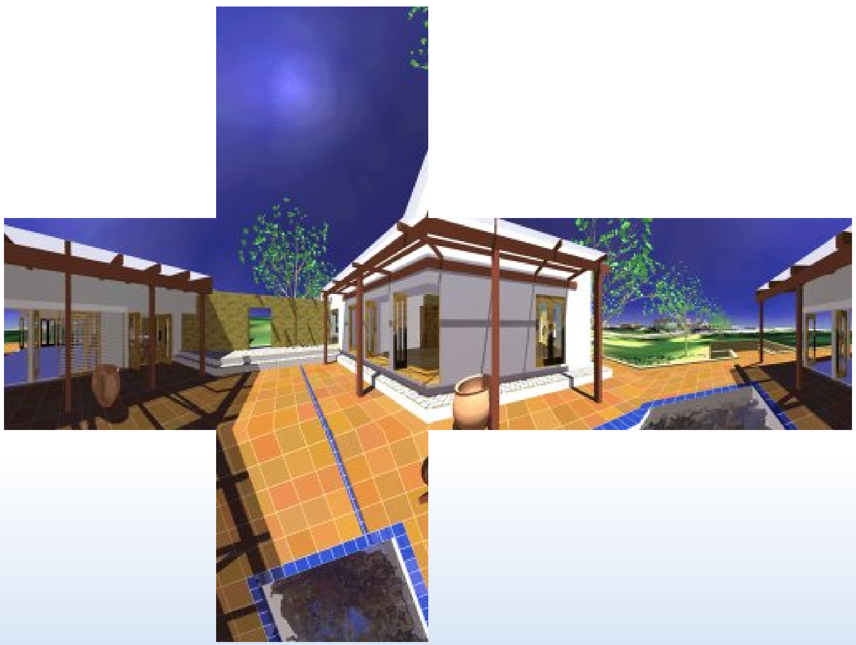
Mapping Techniques



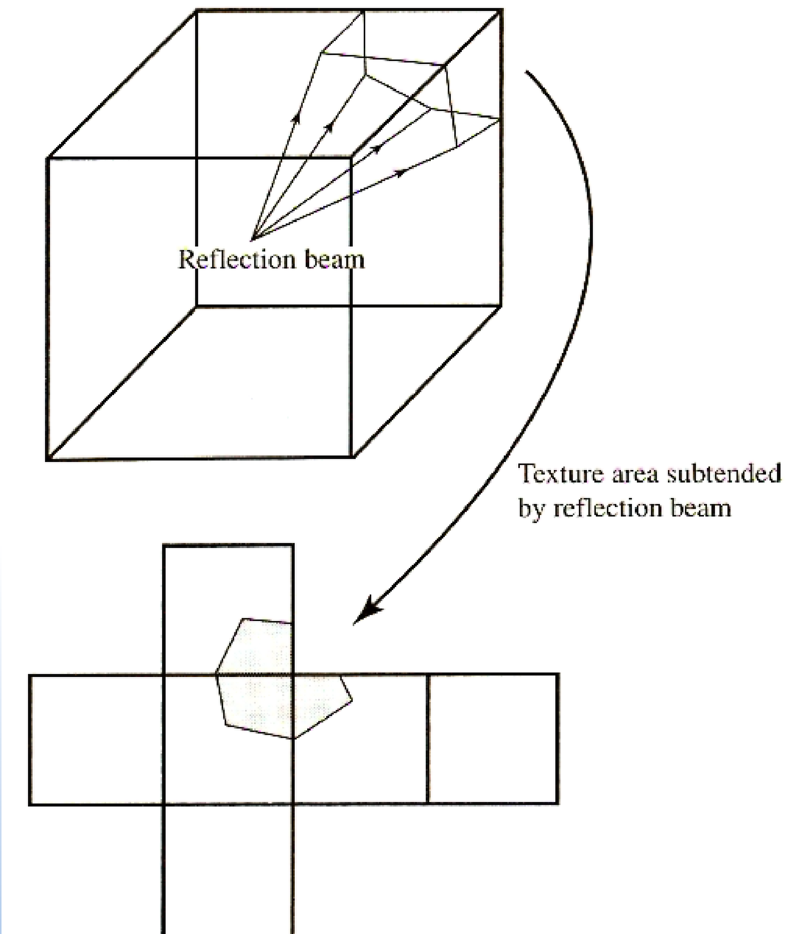
- Cube mapping
  - Popular because easily constructed
    - Place camera in centre of box and render the 6 different views
  - Surrounding scene mapped onto surfaces of a cube

Mapping Techniques

- Environment mapping
  - ➢ Cube mapping

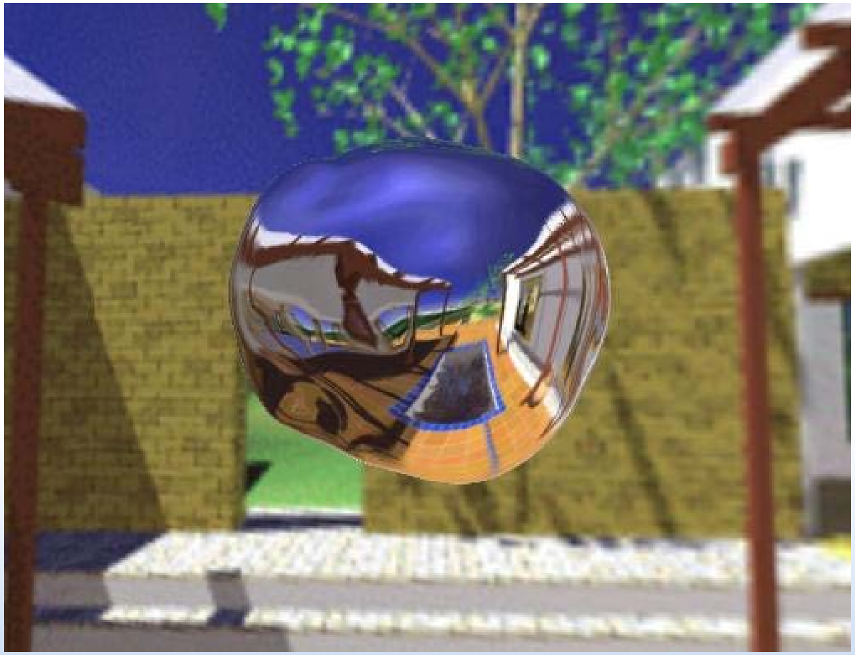


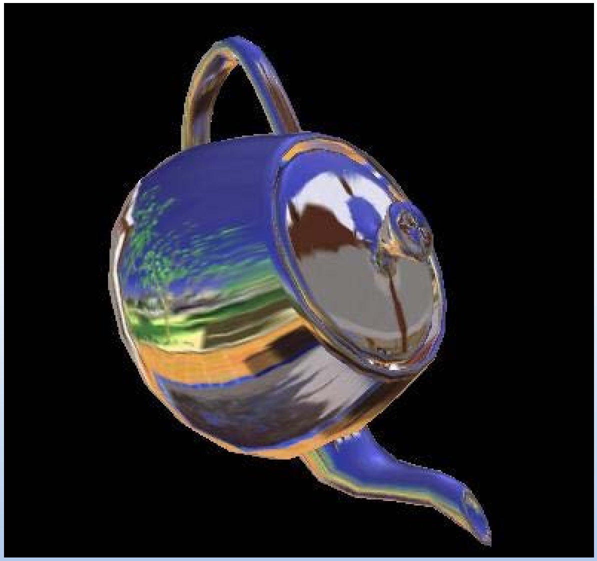
‘Unfolded’ cube



Mapping Techniques

- Environment mapping





Mapping Techniques

- Environment mapping

https://www.farmonline.com.au/story/4761025/self-healing-structures-insect-defenders-and-language-pills-welcome-to-the-future/

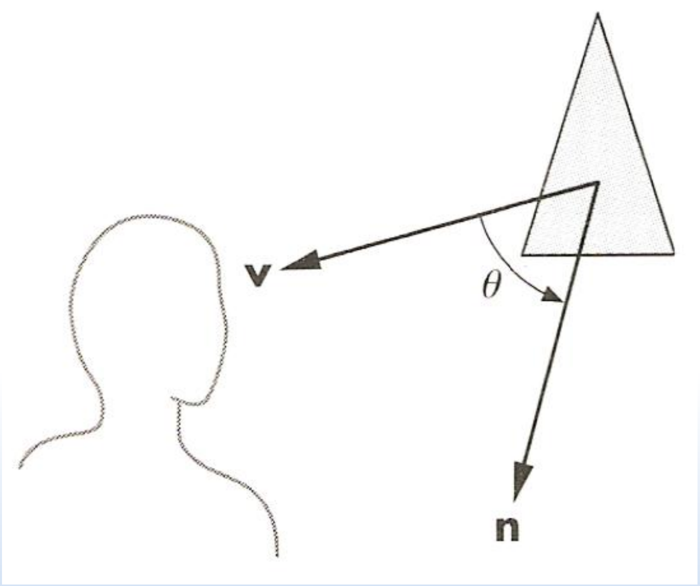
Visibility Determination

- Back-face detection
  - Fast and simple object-space approach for determining back face of a polygon
  - A polygon is facing forward if

    $-90^{\circ} \leq \theta \leq 90^{\circ}$
  - or equivalently

    $\cos\theta \geq 0$
  - Easier to test using dot product instead of cosine

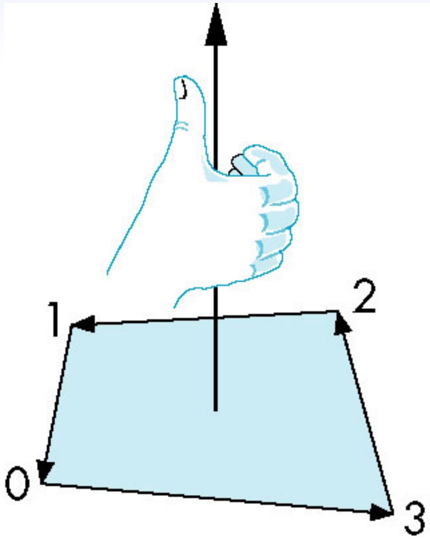
    $\mathbf{n.v} \geq 0$

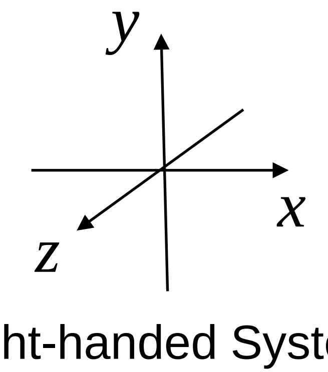


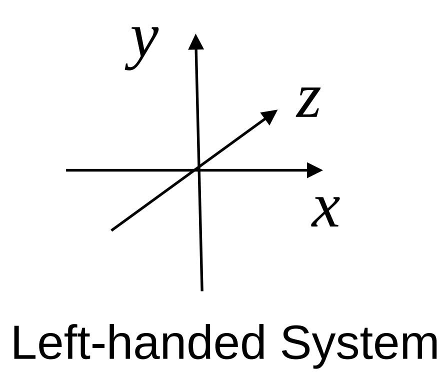
Visibility Determination

- Back-face detection (cont.)
  - Right-hand system
    - Polygon is back face if $C \leq 0$
    - Default cull clockwise (CW) winding
  - Left-hand system
    - Polygon is back face if $C \geq 0$
    - Default cull counter-clockwise (CCW) winding

Polygon Winding

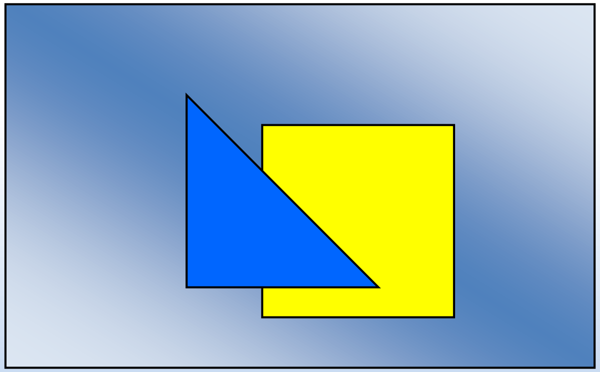






Visibility Determination

- Back-face detection (cont.)
  - ➢ Back-face culling
    - Removal of back-faced polygons
    - Usually happens before the clipping step in a rendering pipeline
    - Does this solve all visibility problems?



No.

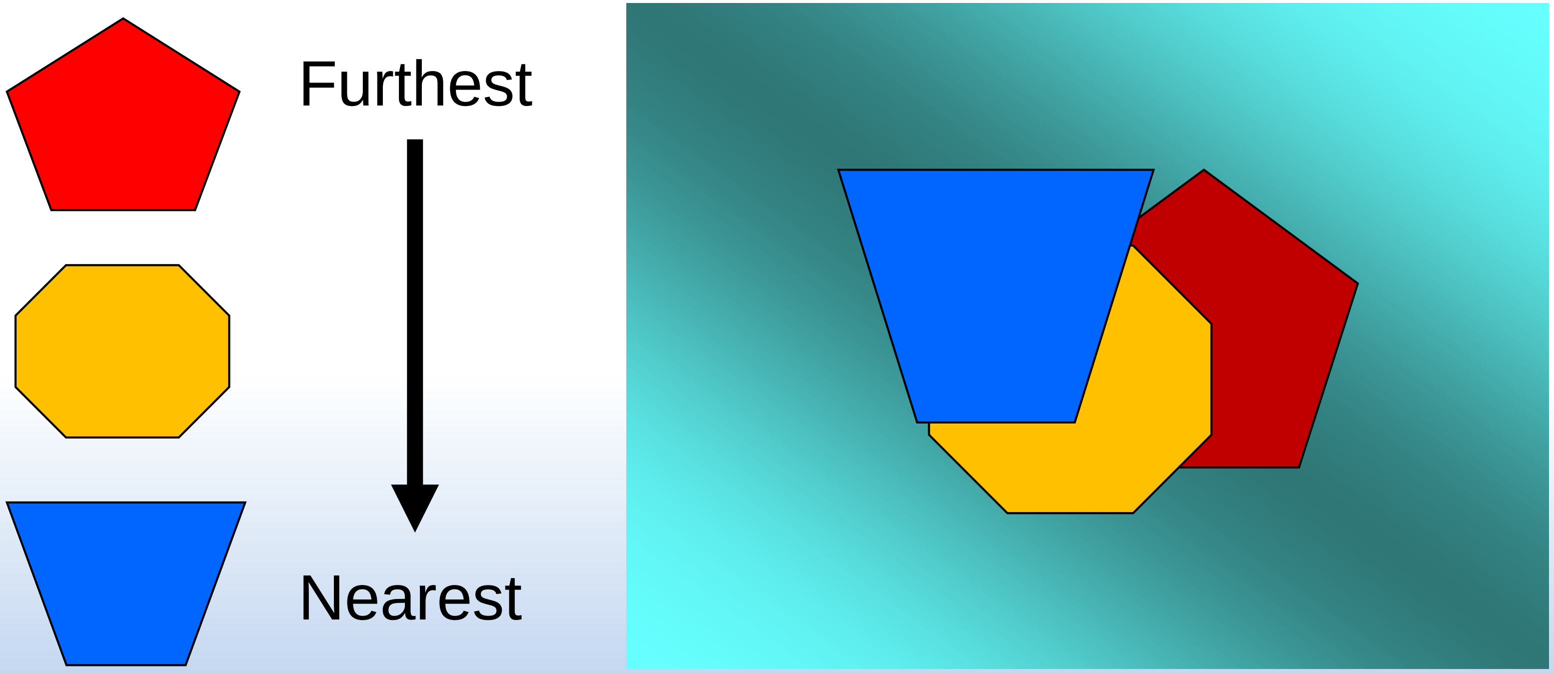
Given these ‘front-facing’ polygons, which sections are visible to the viewer?

Visibility Determination

- The depth-sort method
  - Can be considered to be an object-space approach (the sorting part anyway)
  - General idea
    1. Sort polygons in order of decreasing depth (i.e. furthest to nearest)
    2. Scan convert polygons in order, starting from the furthest (greatest depth) polygon first
  - Nearer polygons will overwrite further polygons
  - This back-to-front rendering often referred to as the 'painter's algorithm'

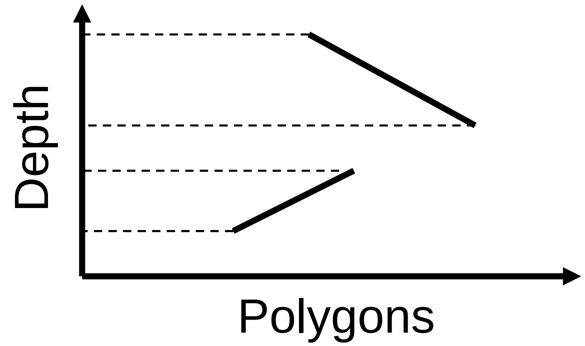
Visibility Determination

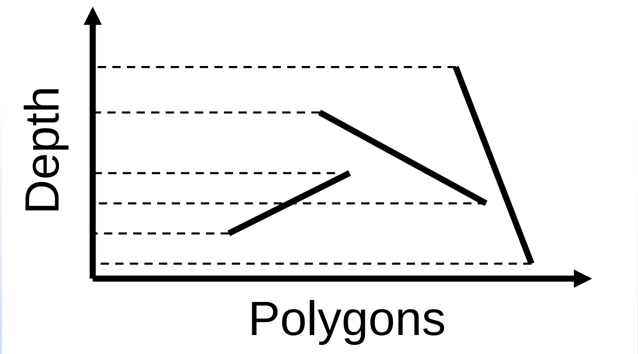
- The depth-sort method (cont.)
  - Obviously, depth sorting has to occur before rasterization



Visibility Determination

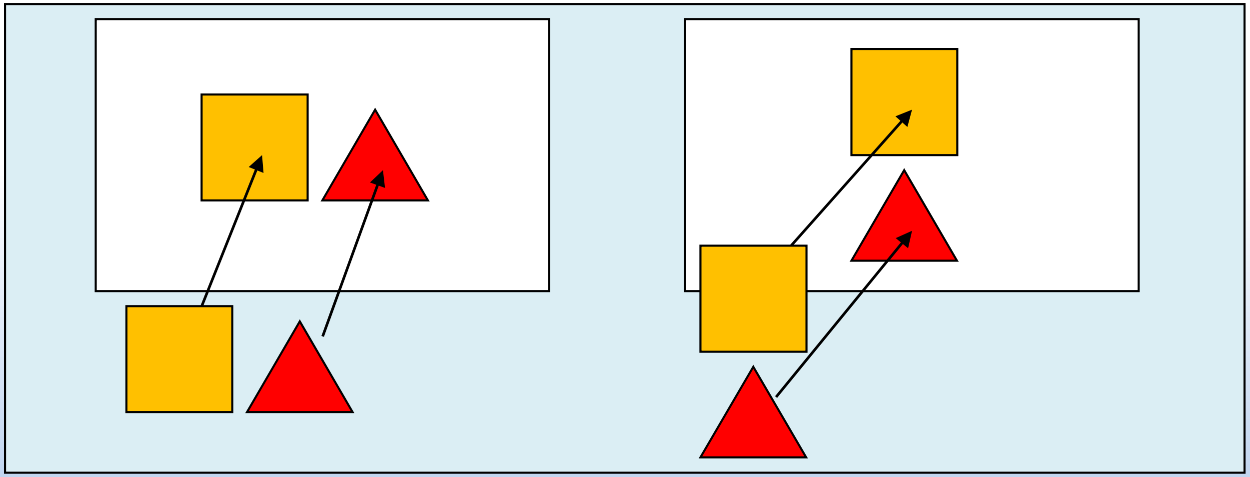
- The depth-sort method (cont.)
  - ➢ This is easy if all the polygons' depths do not overlap
  - ➢ But, how do we sort polygons with overlapping depths?
    - For polygons with overlapping depth, need to make additional comparisons





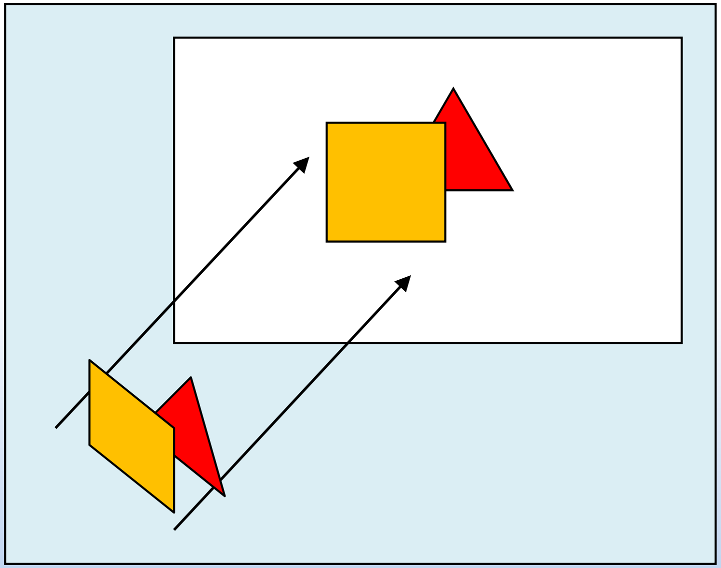
Visibility Determination

- The depth-sort method (cont.)
  - ➢ Check whether the polygons overlap in x and y extents
    - Assuming normalized device coordinates where viewing direction is along z-axis, which is usually the case
    - If non-overlapping x and y, then the order does not matter



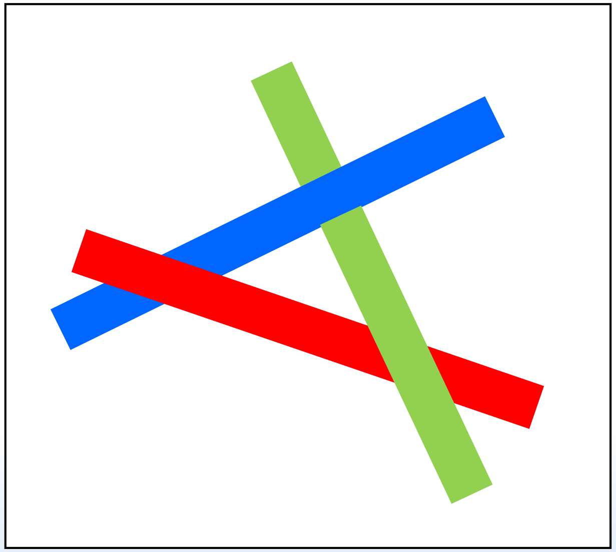
Visibility Determination

- The depth-sort method (cont.)
  - If previous test failed
    - Then test whether the all the vertices of one polygon lie on the same side of the plane defined by the other polygon
    - Order relative to viewing position

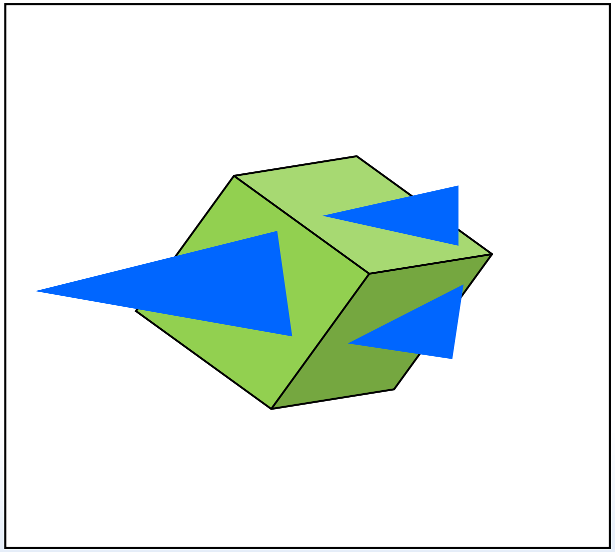


Visibility Determination

- The depth-sort method (cont.)
  - This still leaves two troublesome situations



Cyclic Overlapping



Piercing Polygons

Visibility Determination

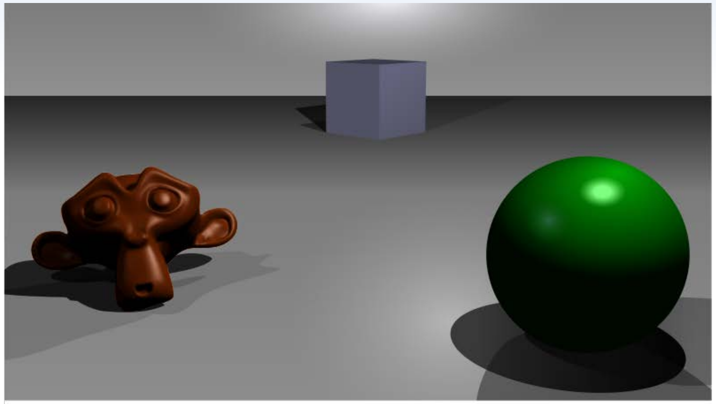
- The depth-sort method (cont.)
  - In such cases, have to resort to splitting polygons
  - Does this solve all visibility problems?

    Yes

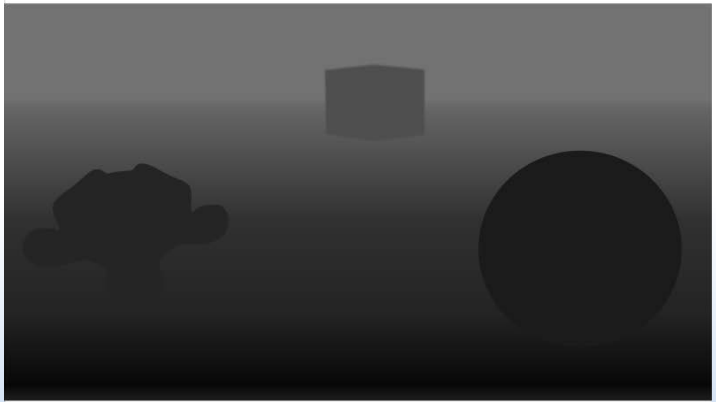
    But depth sorting is somewhat laborious, particularly if we have to split polygons

Visibility Determination

- The depth-buffer method
  - ➢ Commonly used image-space approach
  - ➢ Compares surface depth values for each pixel position on the projection plane
  - ➢ Also referred to as the z-buffer method, since object depth is usually measured along z-axis
  - ➢ Easy to implement, in either hardware or software



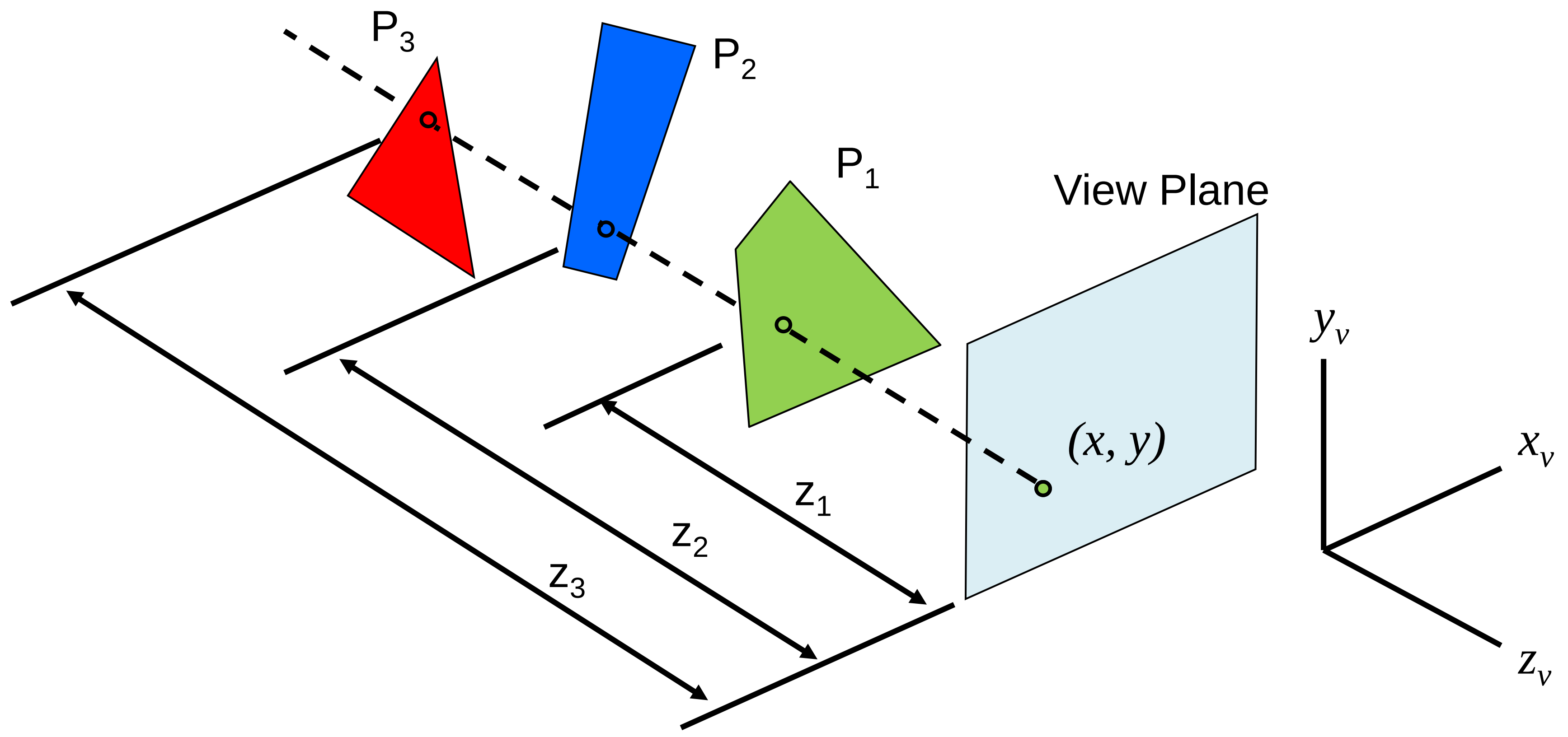
A simple three-dimensional scene



Z-buffer representation

Visibility Determination

- The depth-buffer method (cont.)



Visibility Determination

- The depth-buffer method (cont.)
  - Other than colour buffer
    - Requires another buffer (as the name already implies) to store depth information
  - The depth buffer
    - Has the same resolution as the frame buffer and stores floating point numbers
    - Typically carried out in normalized coordinates, so depth values range from 0.0 (near clipping plane) to 1.0 (far clipping plane)

Visibility Determination

- The depth-buffer method (cont.)
  1. Initialise depth buffer and colour buffer for all positions

     depthBuff (x, y) = 1.0f

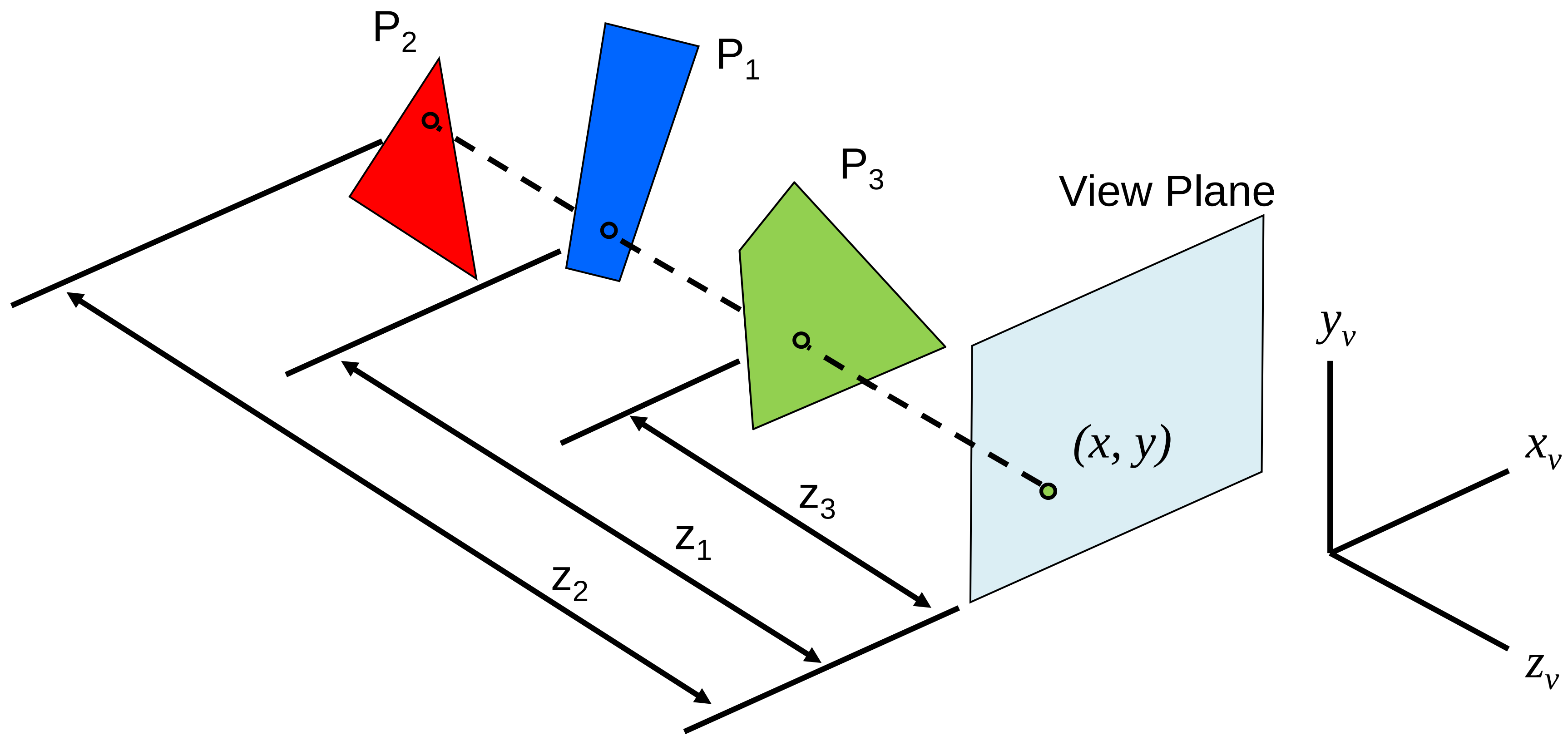
     colourBuff (x, y) = backgroundColour
  2. For each polygon in the scene (in any order)
     - For each projected (x, y) pixel position of polygon, calculate the depth z (if not already known)
     - If z < depthBuff (x, y), then compute and set the surface colour at that position

       depthBuff (x, y) = z

       colourBuff (x, y) = surfColour (x, y)

Visibility Determination

- The depth-buffer method (cont.)



Visibility Determination

- The depth-buffer method (cont.)
  - Polygons can be processed in one at a time in any order and each polygon is processed one scan line at a time
  - Compatible with the pipeline architecture as it is performed polygon by polygon
  - Does this solve all visibility problems?

    Yes

    Used to be a problem when memory was expensive

References

- Among others, material sourced from
  - Hearn, Baker & Carithers, “Computer Graphics with OpenGL”, Prentice-Hall
  - Angel & Shreiner, “Interactive Computer Graphics: A Top-Down Approach with OpenGL”, Addison Wesley
  - Akenine-Moller, Haines & Hoffman, “Real Time Rendering”, A.K. Peters
  - Joey de Vries, “Learn OpenGL,” https://learnopengl.com/
  - Watt, “3D Computer Graphics”, Pearson/Addison Wesley
  - http://en.wikipedia.org/wiki/
  - IT OpenCourseWare, https://ocw.mit.edu/
  - http://http.developer.nvidia.com/GPUGems/gpugems_ch21.html