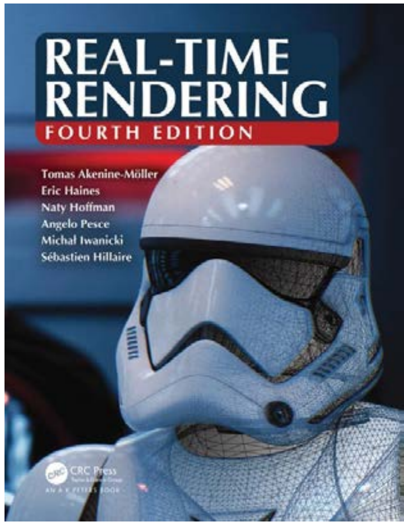
TDA362 – Computer Graphics



Teacher: Ulf Assarsson

Chalmers University of Technology

This Course

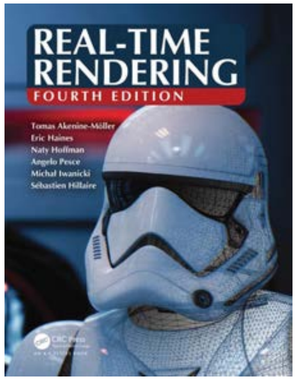
- Algorithms!

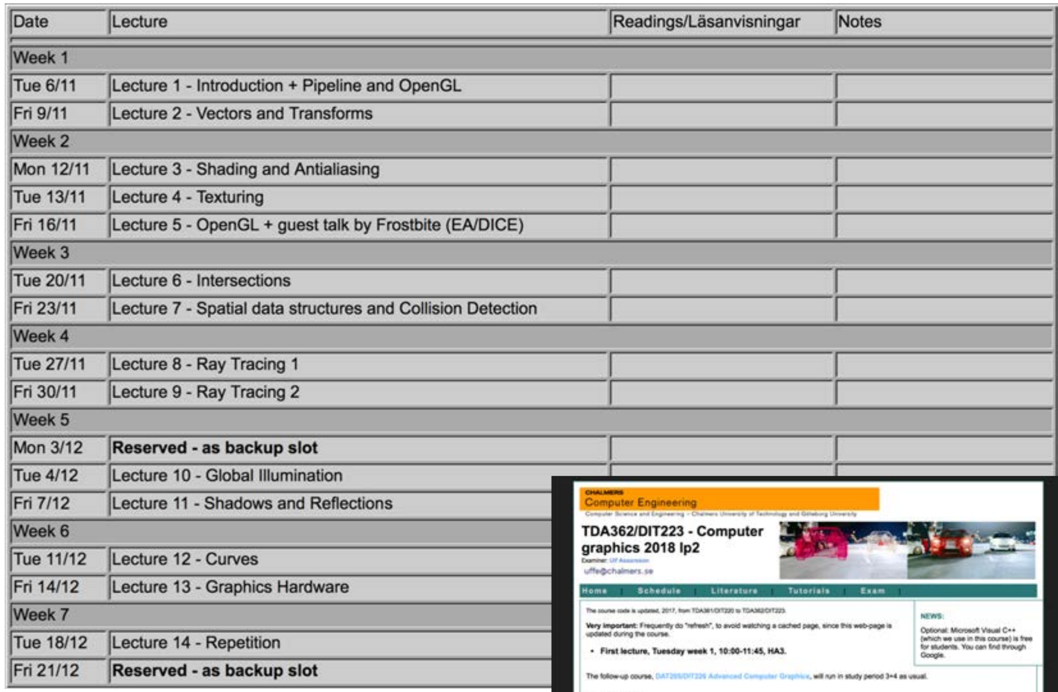
Real-time Rendering

Understanding of Ray Tracing

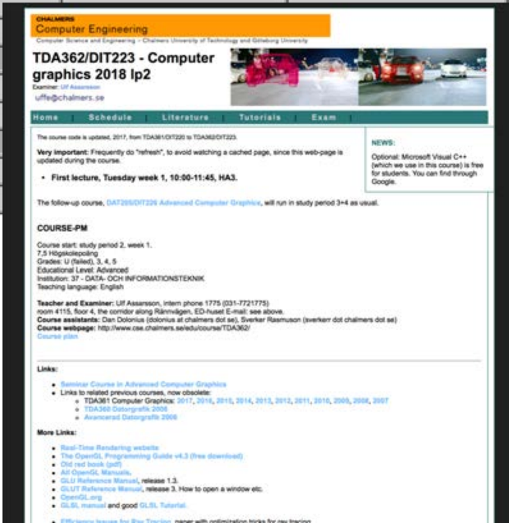
Course Info

- Study Period 2 (lp2)
- Real Time Rendering, 4th edition
    - Available on Cremona at discount.
- Schedule:
    - Mon 13-15, w2 only
    - Tues 10-12,
    - Fri 9-12,
        - ~14 lectures in total, ~2 / week
    - Labs:
        - Mon: 17-21
        - Tues: 13-21
        - Wed: 13-21
        - Thur: 9-12 + 17-21
- Homepage:
    - Google "TDA362" or
    - "Computer Graphics Chalmers"





Date	Lecture	Readings/Läsanvisningar	Notes
Week 1			
Tue 6/11	Lecture 1 - Introduction + Pipeline and OpenGL		
Fri 9/11	Lecture 2 - Vectors and Transforms		
Week 2			
Mon 12/11	Lecture 3 - Shading and Antialiasing		
Tue 13/11	Lecture 4 - Texturing		
Fri 16/11	Lecture 5 - OpenGL + guest talk by Frostbite (EA/DICE)		
Week 3			
Tue 20/11	Lecture 6 - Intersections		
Fri 23/11	Lecture 7 - Spatial data structures and Collision Detection		
Week 4			
Tue 27/11	Lecture 8 - Ray Tracing 1		
Fri 30/11	Lecture 9 - Ray Tracing 2		
Week 5			
Mon 3/12	**Reserved - as backup slot**		
Tue 4/12	Lecture 10 - Global Illumination		
Fri 7/12	Lecture 11 - Shadows and Reflections		
Week 6			
Tue 11/12	Lecture 12 - Curves		
Fri 14/12	Lecture 13 - Graphics Hardware		
Week 7			
Tue 18/12	Lecture 14 - Repetition		
Fri 21/12	**Reserved - as backup slot**		



CHALMERS
Computer Engineering
Computer Science and Engineering - Chalmers University of Technology and Göteborg University

TDA362/DIT223 - Computer graphics 2018 lp2
Examiner: Ulf Assarsson
uffe@chalmers.se

Home | Schedule | Literature | Tutorials | Exam |

The course code is updated, 2017, from TDA361/DIT220 to TDA362/DIT223.

Very important: Frequently do "refresh", to avoid watching a cached page, since this web-page is updated during the course.

- **First lecture, Tuesday week 1, 10:00-11:45, HA3.**

NEWS:
Optional: Microsoft Visual C++ (which we use in this course) is free for students. You can find through Google.

The follow-up course, DAT205/DIT226 Advanced Computer Graphics, will run in study period 3+4 as usual.

COURSE-PM

Course start: study period 2, week 1.
7,5 Högskolepoäng
Grades: U (failed), 3, 4, 5
Educational Level: Advanced
Institution: 37 - DATA- OCH INFORMATIONSTEKNIK
Teaching language: English

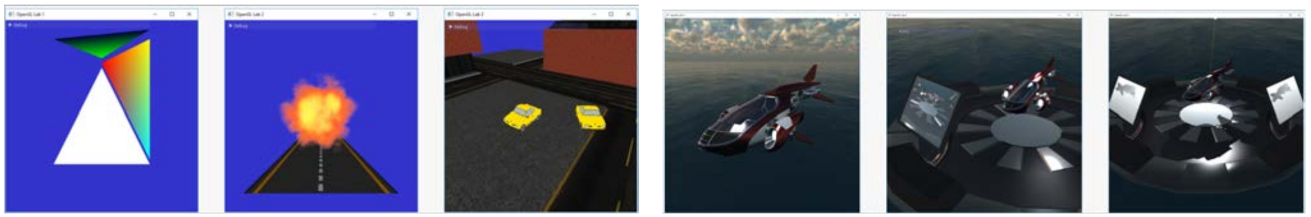
Teacher and Examiner: Ulf Assarsson, intern phone 1775 (031-7721775)
room 4115, floor 4, the corridor along Rännvägen, ED-huset E-mail: see above.
Course assistants: Dan Dolonius (dolonius at chalmers dot se), Sverker Rasmuson (sverkerr dot chalmers dot se)
Course webpage: http://www.cse.chalmers.se/edu/course/TDA362/
Course plan

Links:

- Seminar Course in Advanced Computer Graphics
- Links to related previous courses, now obsolete:
    - TDA361 Computer Graphics: 2017, 2016, 2015, 2014, 2013, 2012, 2011, 2010, 2009, 2008, 2007
    - TDA360 Datorgrafik 2006
    - Avancerad Datorgrafik 2006

More Links:

- Real-Time Rendering website
- The OpenGL Programming Guide v4.3 (free download)
- Old red book (pdf)
- All OpenGL Manuals,
- GLU Reference Manual, release 1.3.
- GLUT Reference Manual, release 3. How to open a window etc.
- OpenGL.org
- GLSL manual and good GLSL Tutorial.
- Efficiency Issues for Ray Tracing, paper with optimization tricks for ray tracing

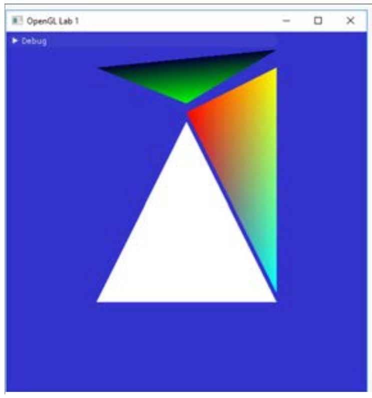


Tutorials

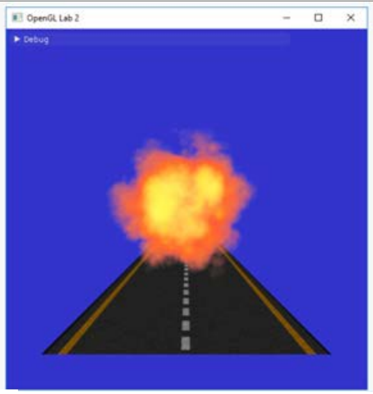
- All laborations are in C++ and OpenGL
  - Industry standard
  - No previous (C++) knowledge required
- Six shorter tutorials that go through basic concepts
  - Basics, Textures, Camera&Animation, Shading, Render-to-texture, Shadow Mapping
- One slightly longer lab where you put everything together
  - Real-time rendering

    or
  - Path tracer

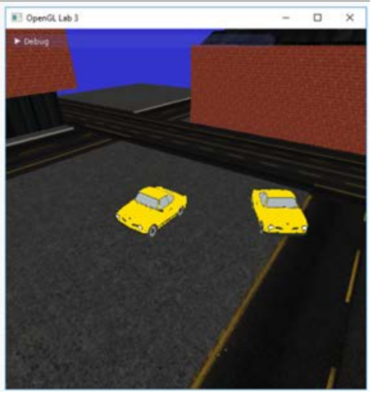
Tutorials 1-6



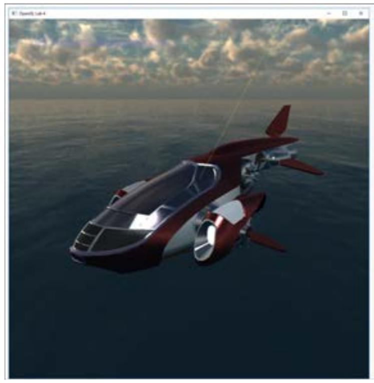
Rendering a triangle



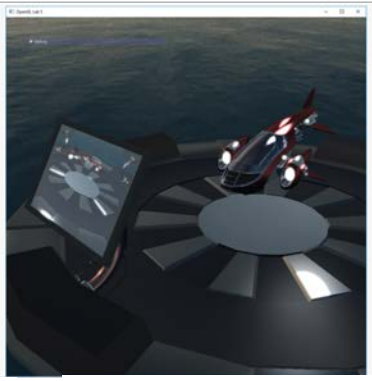
Textures



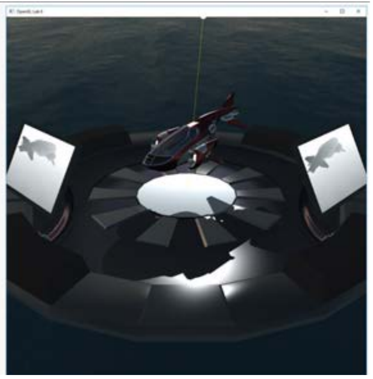
Animation



Shading



Render to textures

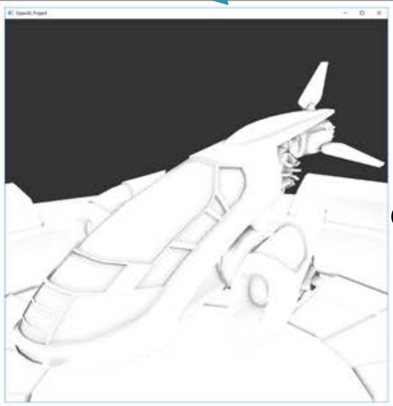


Shadow maps

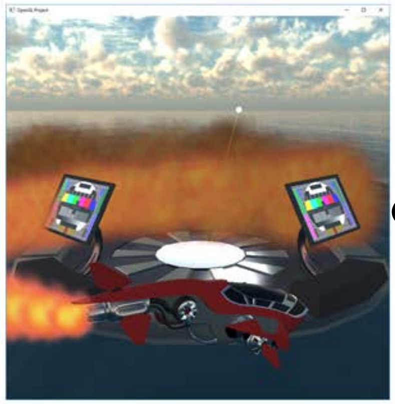
Project

Choose at least 1 from:

$$\begin{tikzcd} & \text{Project} \arrow[dl] \arrow[dr] & \\ \text{Real-time rendering} & & \text{Offline rendering} \end{tikzcd}$$


Screen-space ambient occlusion

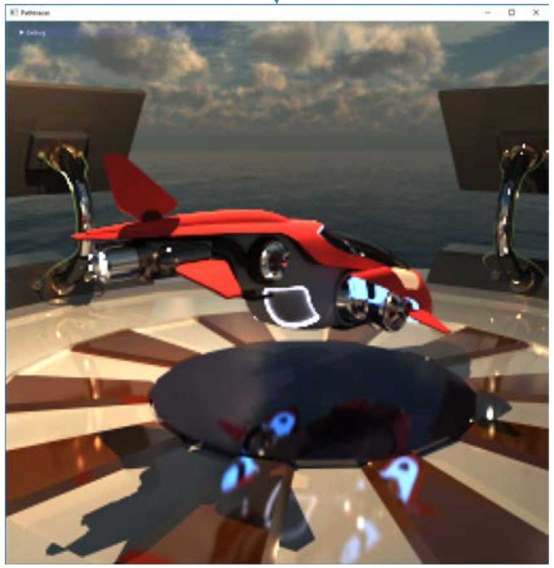
or


Particle System

or

Height field

or

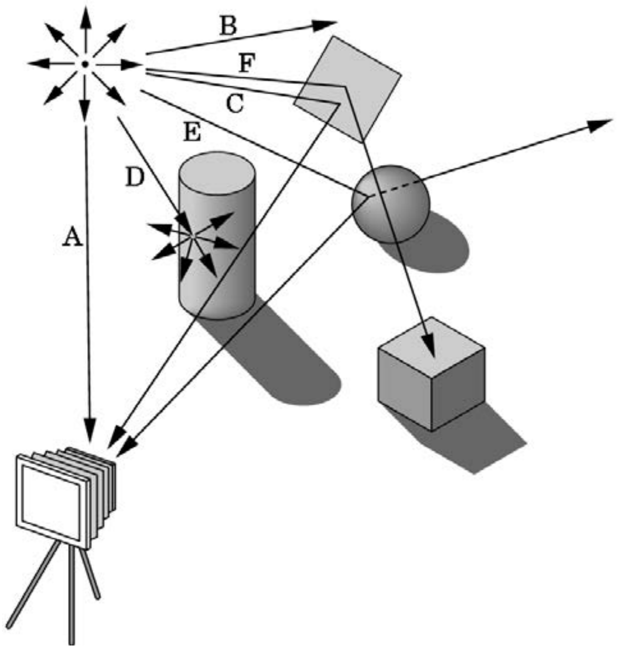

Path Tracing

Tutorials

- Info: http://www.cse.chalmers.se/edu/course/TDA362/tutorials.html
- Rooms 4225, 4th floor EDIT-building
  - Or your favorite place/home
- Time slots available every day.
  - No booking. First come first served.
- To pass the tutorials:
  - Show your solutions to lab assistants in lab rooms (bring your computer if done at home)
  - Deadlines:
    - Lab 1+2+3: Thursday week 2.
    - Lab 4 + 5: Thursday week 3.
    - Lab 6: Thursday week 4
    - Lab 7 / Project: Thursday week 7.
- Do the tutorials in groups (Labgrupper) of two, or individually if you prefer.
  - If you want a lab partner
    - Write your name + email on list at desk in the break

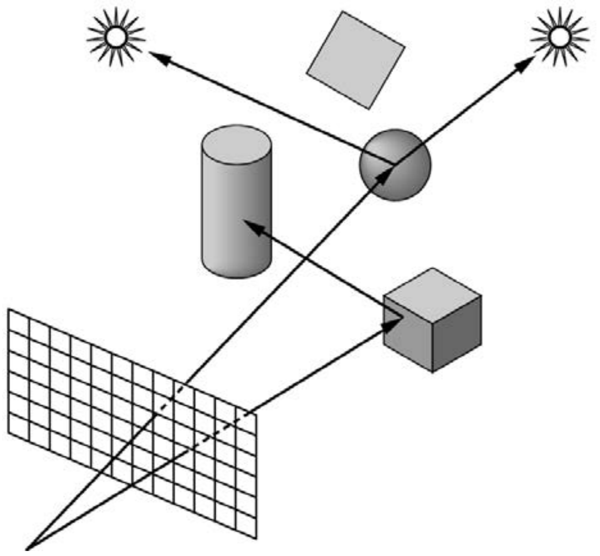
Tracing Photons

One way to form an image is to follow rays of light from a point source finding which rays enter the lens of the camera. However, each ray of light may have multiple interactions with objects before being absorbed or going to infinity.



Other Physical Approaches

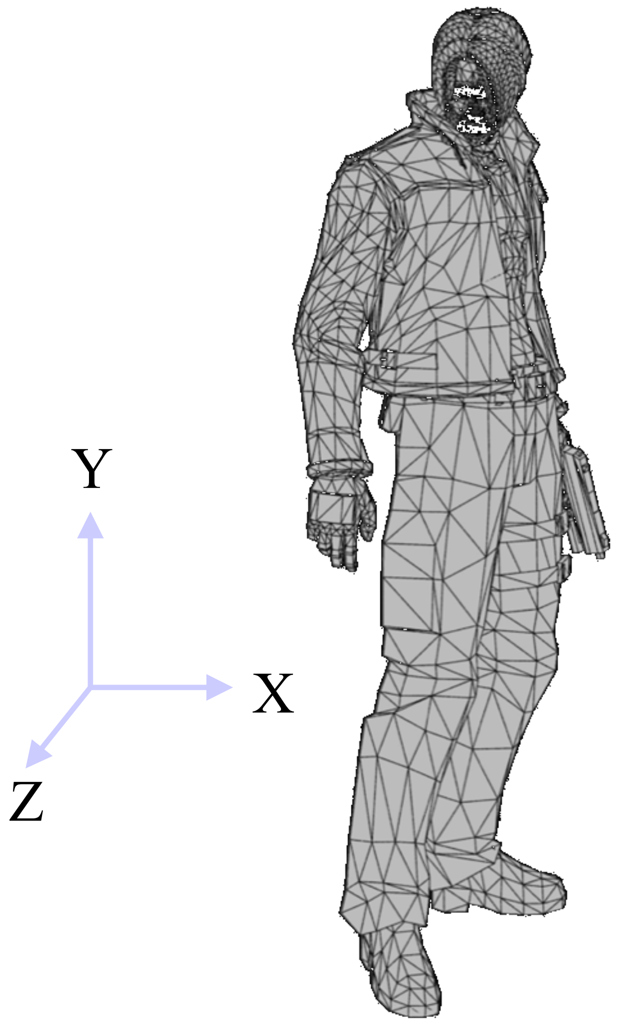
- **Ray tracing**: follow rays of light from center of projection until they either are absorbed by objects or go off to infinity
  - Can handle global effects
    - Multiple reflections
    - Translucent objects
  - Faster but still rather slow

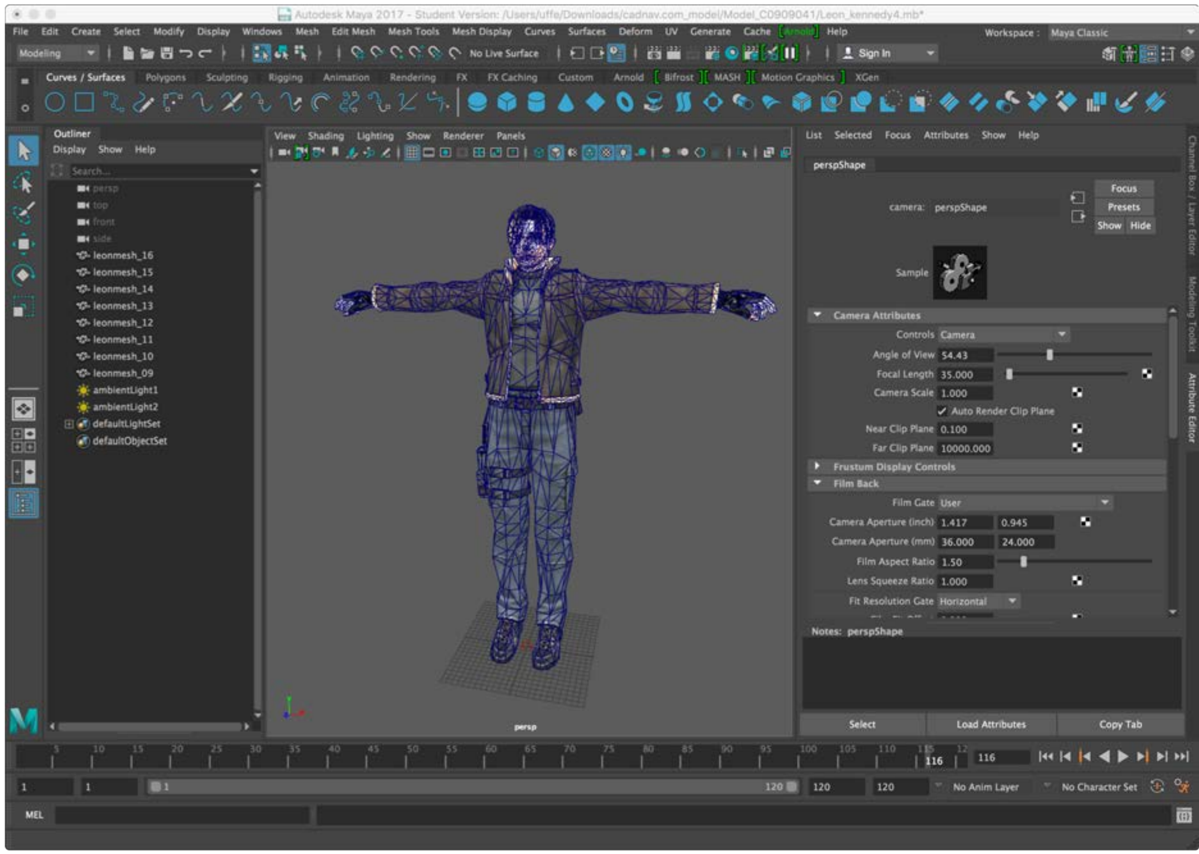


Real-Time Rendering

Overview of the Graphics Rendering Pipeline and OpenGL

3D-models: surfaces are constructed by triangles.





camera: perspShape

Controls: Camera
Angle of View: 54.43
Focal Length: 35.000
Camera Scale: 1.000
✓ Auto Render Clip Plane
Near Clip Plane: 0.100
Far Clip Plane: 10000.000

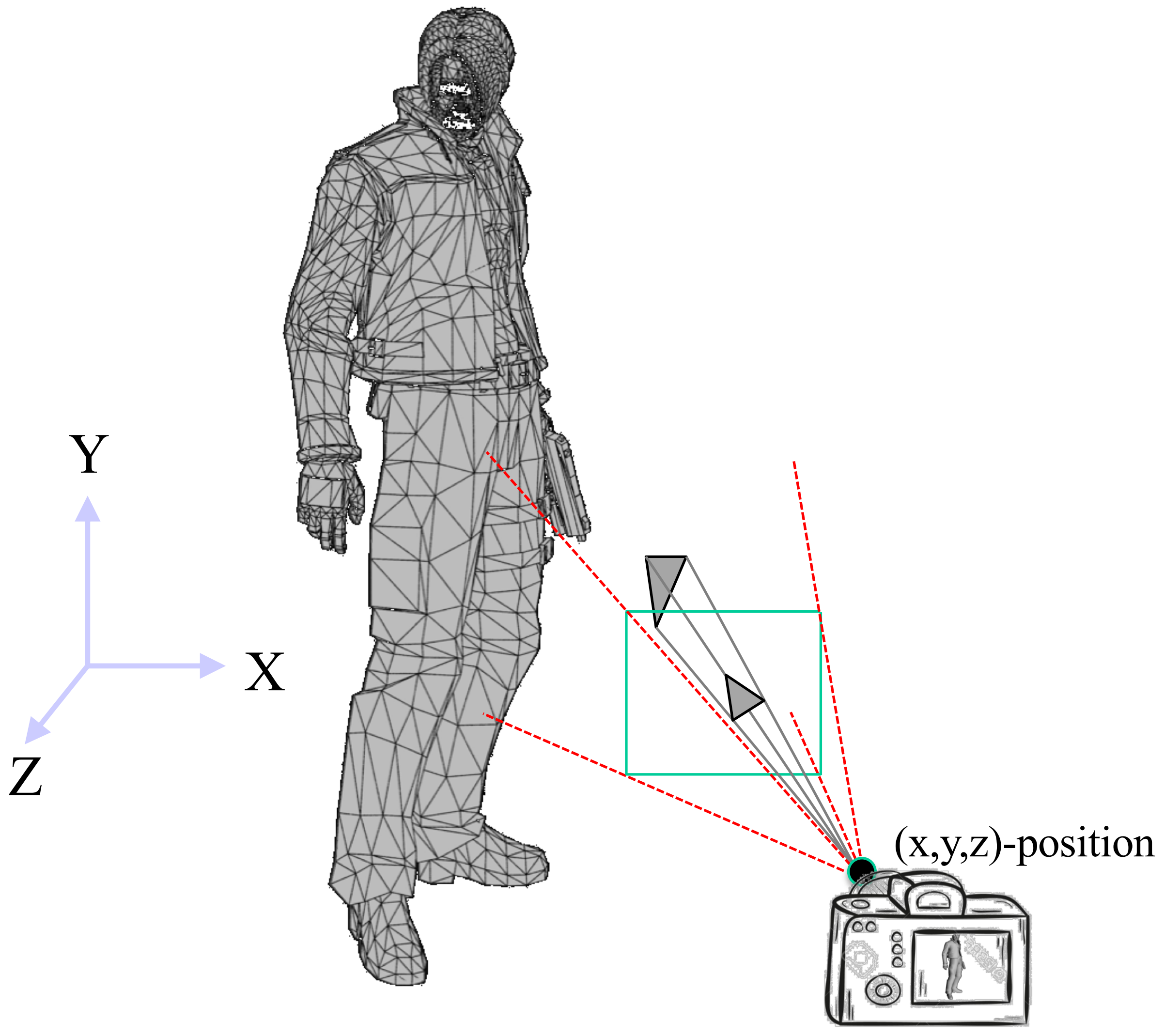
Film Gate: User
Camera Aperture (inch): 1.417 0.945
Camera Aperture (mm): 36.000 24.000
Film Aspect Ratio: 1.50
Lens Squeeze Ratio: 1.000
Fit Resolution Gate: Horizontal

Notes: perspShape

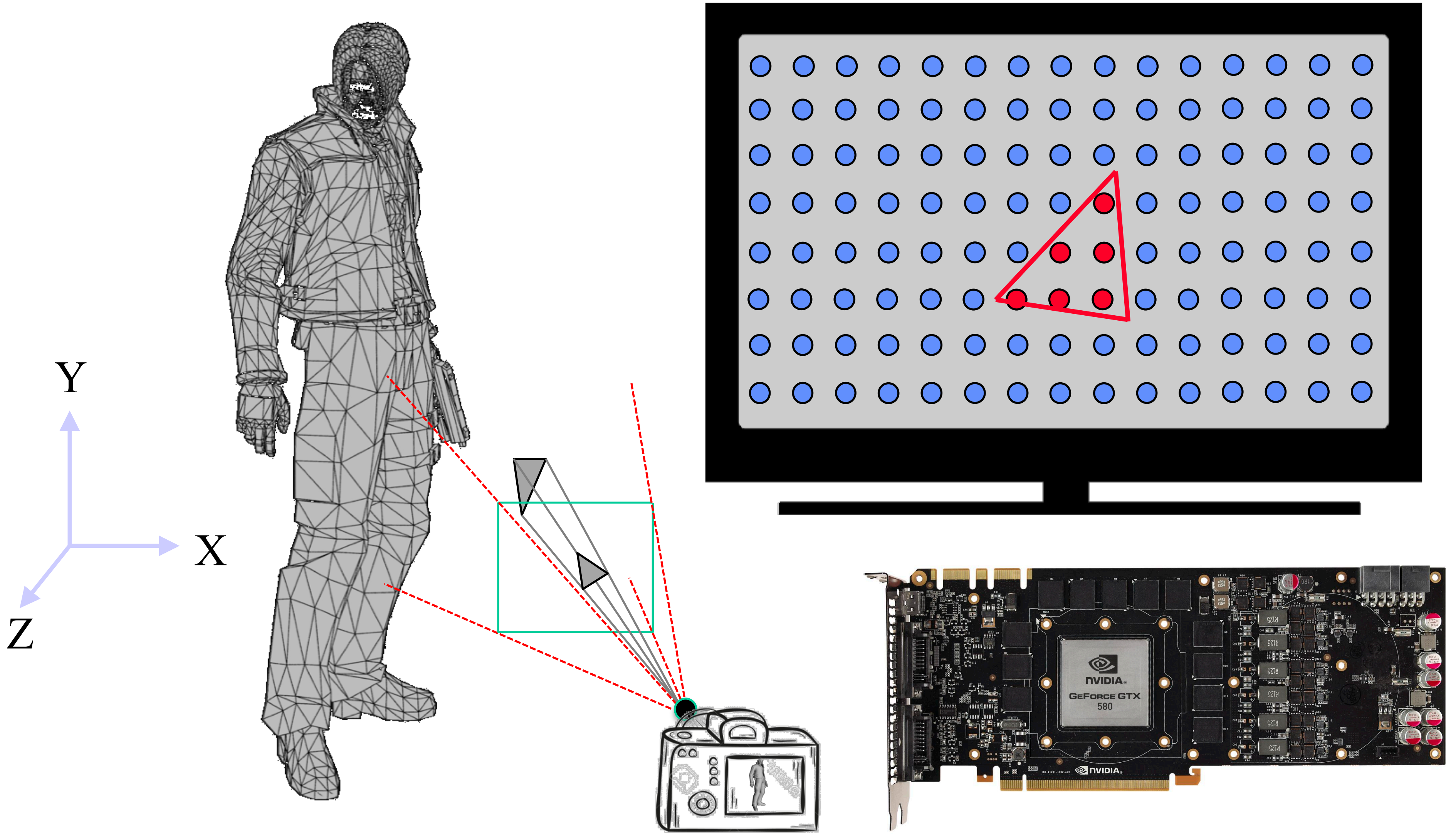
4926 triangles

Why triangles?

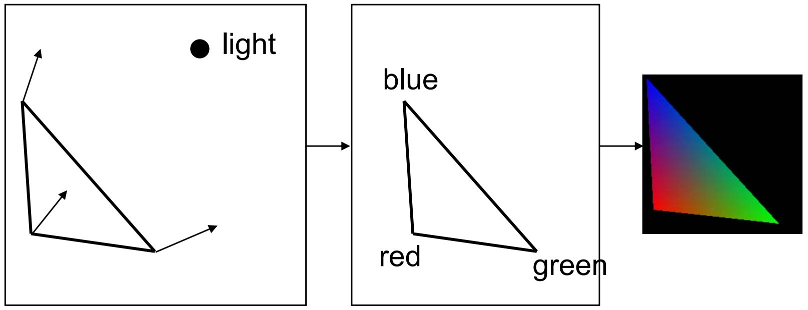
Each triangle is projected onto the image plane using a virtual camera.



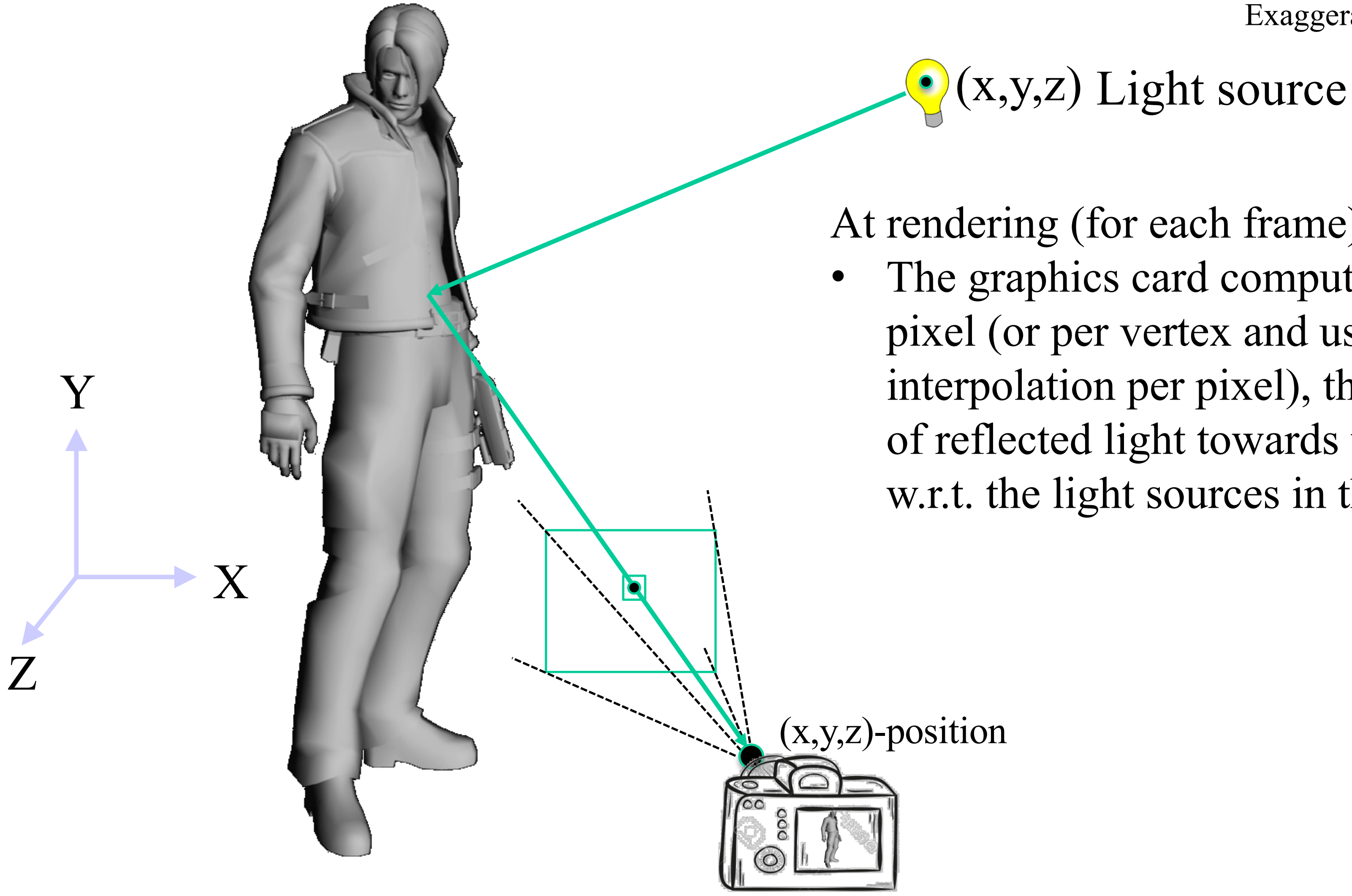
The graphics card draws the triangles onto the screen.



E.g., adding lighting



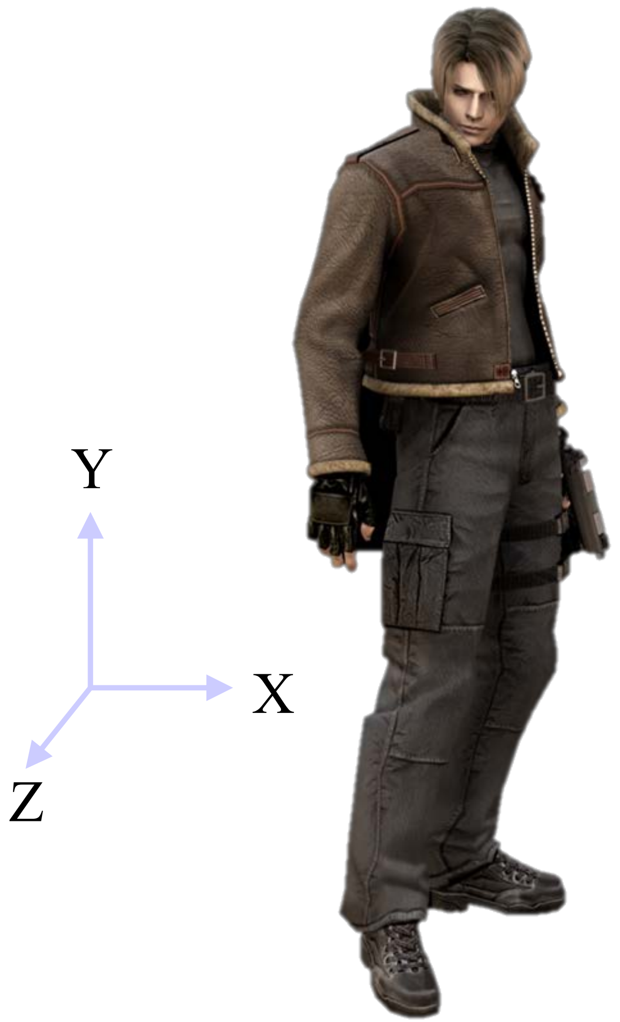
Exaggerated example



At rendering (for each frame):

- The graphics card computes, per pixel (or per vertex and using interpolation per pixel), the amount of reflected light towards the camera, w.r.t. the light sources in the scene.

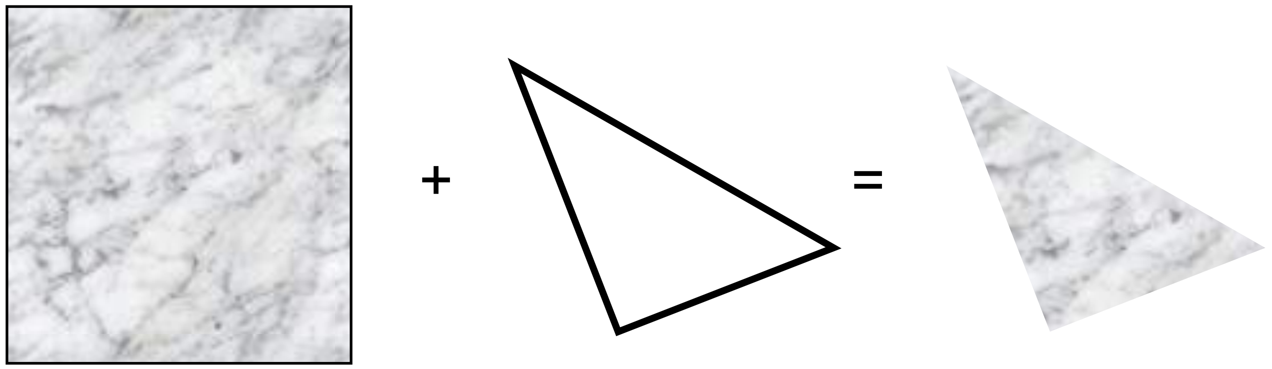
E.g., adding textures – to simulate details and materials



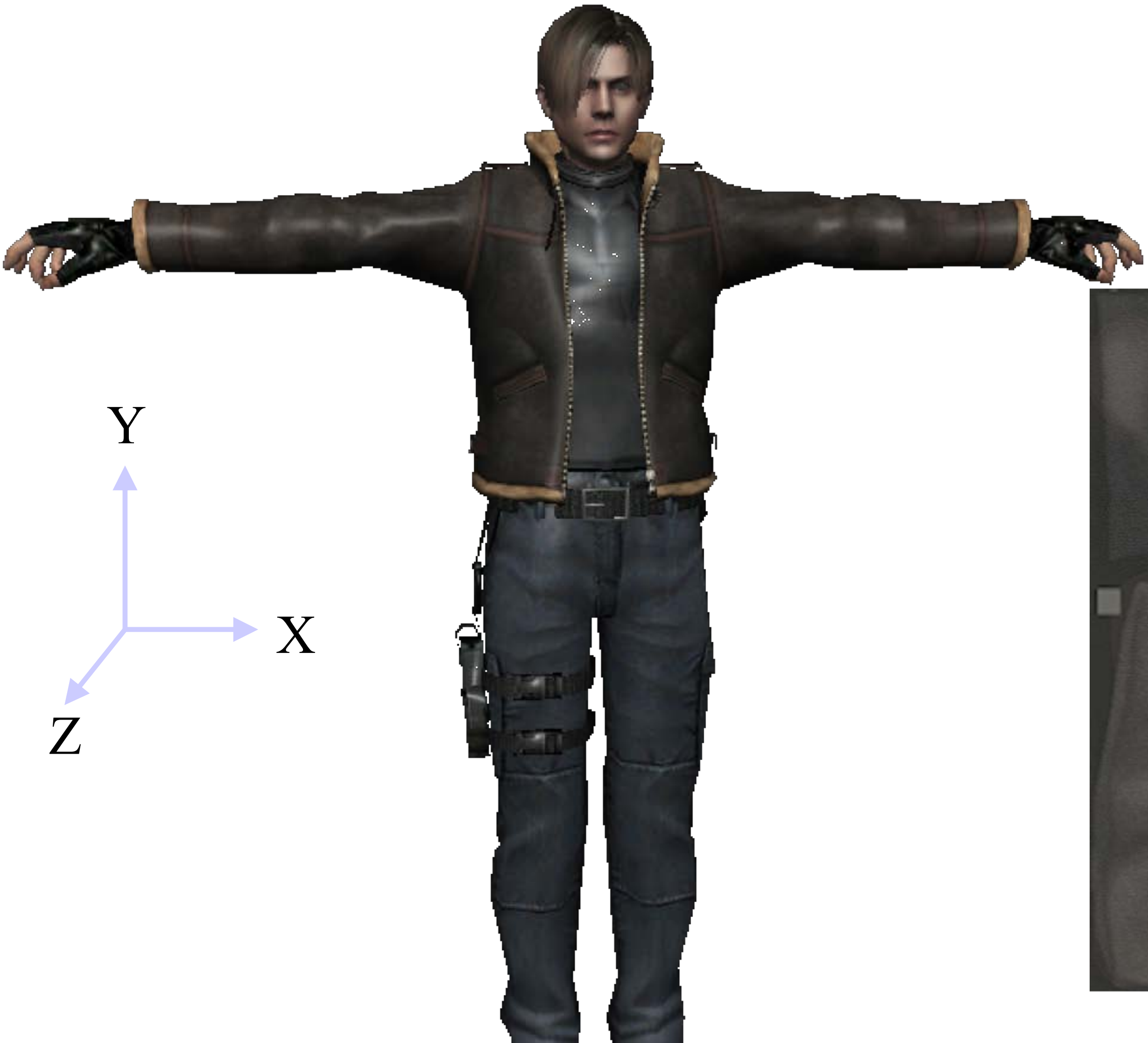
At rendering (for each frame):

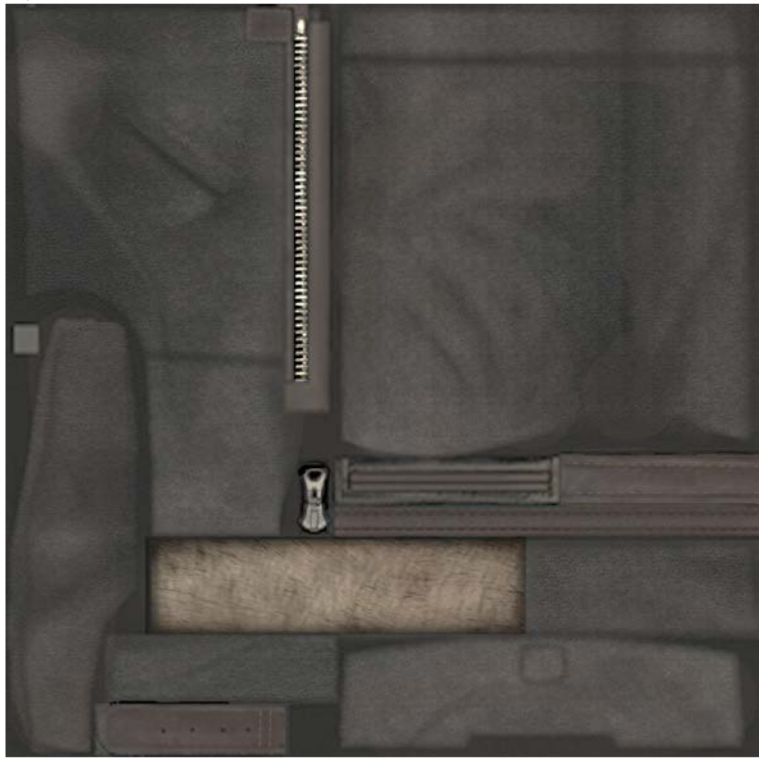
- The graphics card computes, per pixel (or per vertex and using interpolation per pixel), the amount of reflected light towards the camera, w.r.t. the light sources in the scene.
- And adds **textures** (=images) on the triangles (modulated with the light intensity) to simulate surface details and different materials.

Adding textures

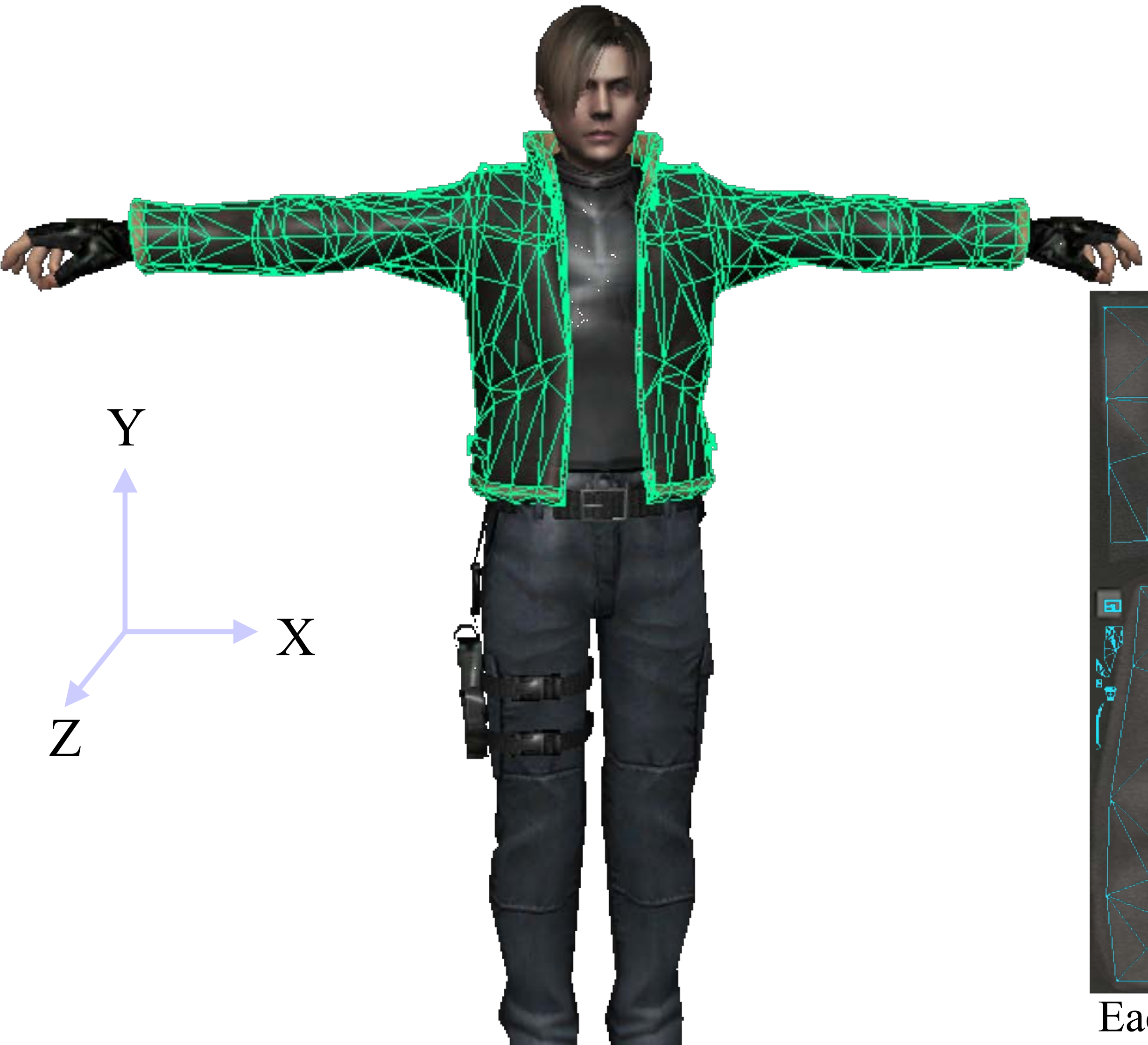


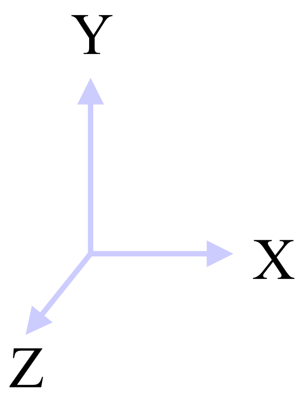
Textures (a.k.a. Texture Maps)

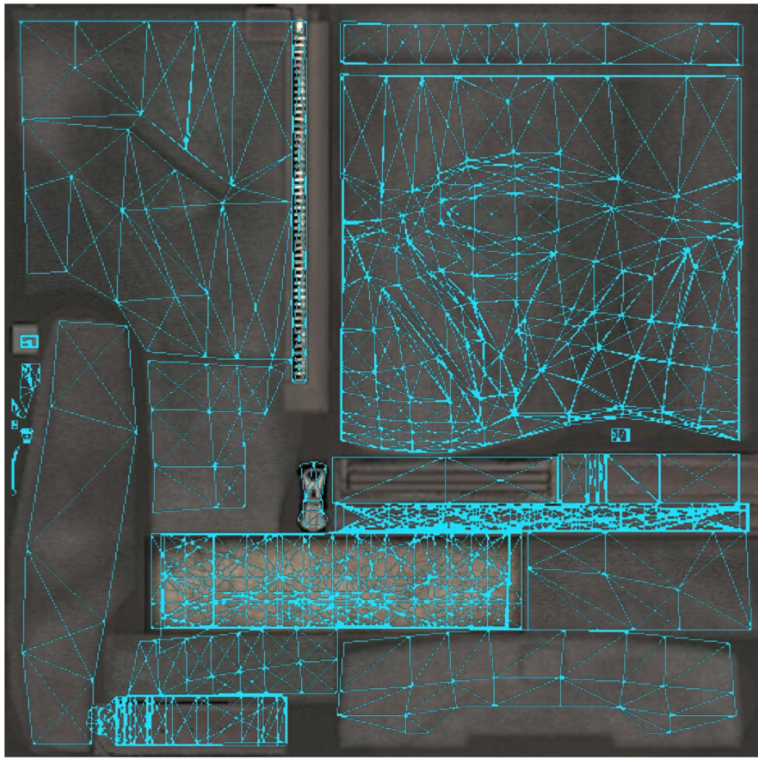




Textures (a.k.a. Texture Maps)

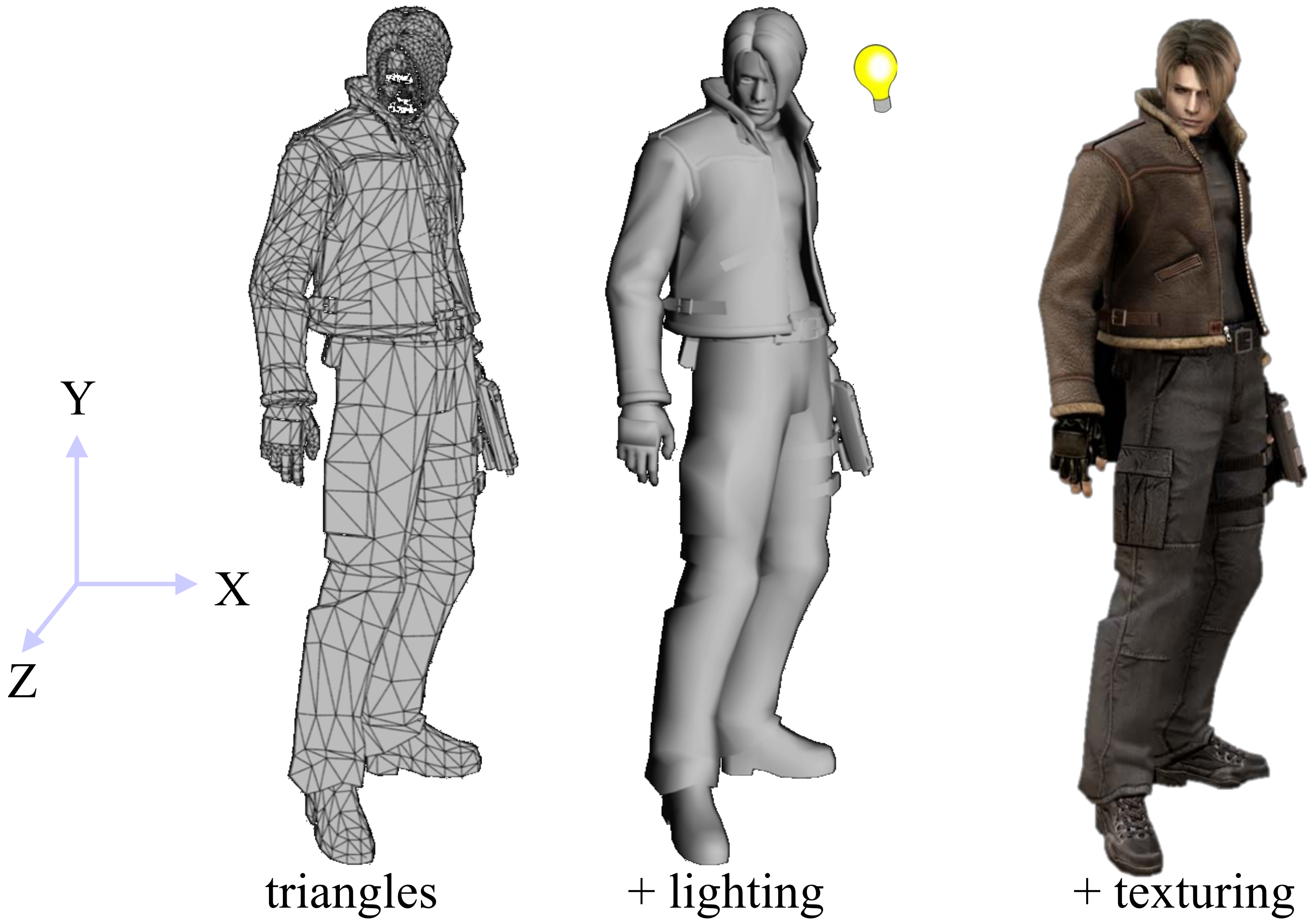






Each triangle's layout in texture space

Summary of this very common type of appearance modeling.

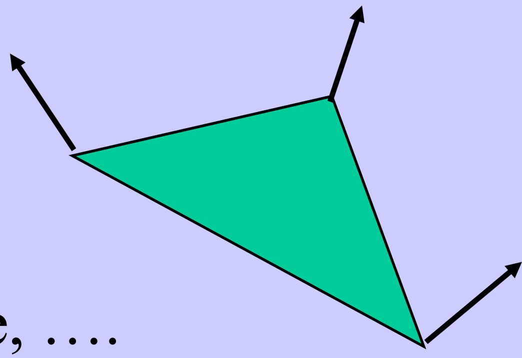


The Graphics Rendering Pipeline

The Application stage, geometry stage, and rasterizer stage

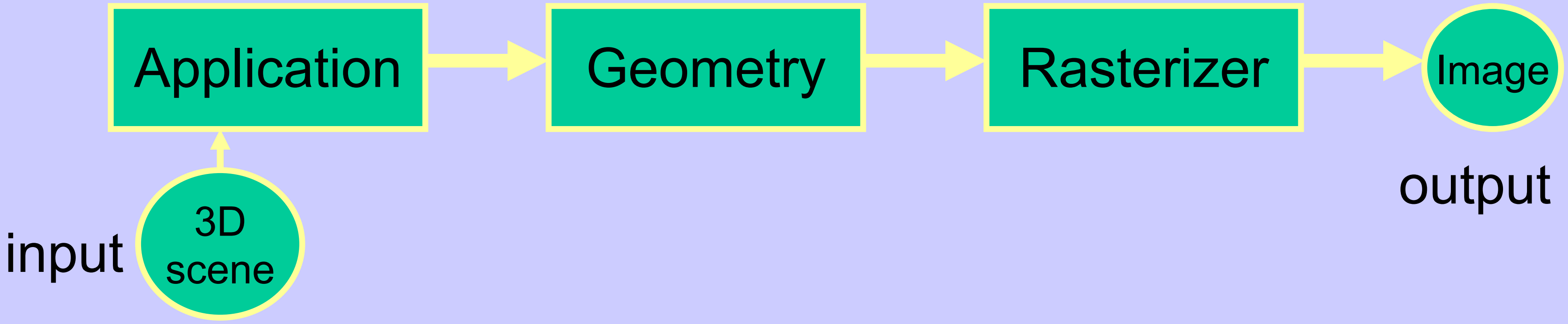
You say that you render a ”*3D scene*”, but what is it?

- First, of all to take a picture, it takes a camera – a virtual one.
  - Decides what should end up in the final image
- A 3D scene is:
  - Geometry (triangles, lines, points, and more)
  - Light sources
  - Material properties of geometry
    - Colors, shader code ,
    - Textures (images to glue onto the geometry)
- A triangle consists of 3 vertices
  - A vertex is 3D position, and may include a normal, color, texture coordinate, ….



Lecture 1: Real-time Rendering The Graphics Rendering Pipeline

- The pipeline is the ”engine” that creates images from 3D scenes
- Three conceptual stages of the pipeline:
  - Application (executed on the CPU)
  - Geometry
  - Rasterizer



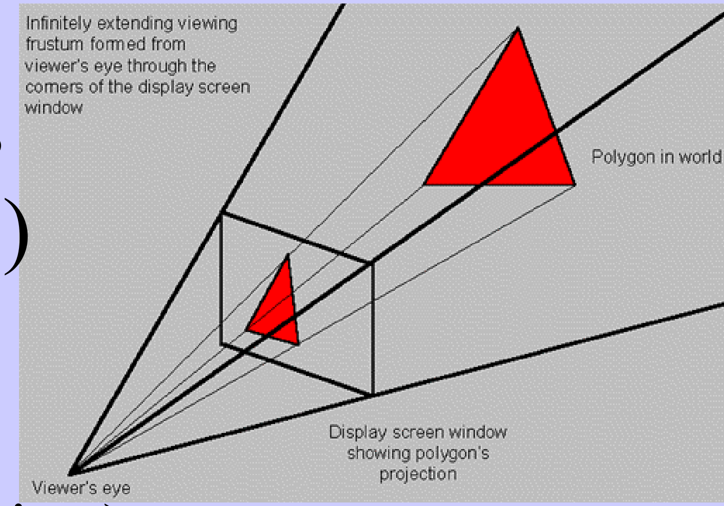
Application
Geometry
Rasterizer

The APPLICATION stage

- Executed on the CPU
  - Means that the programmer decides what happens here
- Examples:
  - Collision detection
  - Speed-up techniques
  - Animation
- Most important task: feed geometry stage with the primitives (e.g. triangles) to render

$$\begin{tikzcd}\text{Application} \arrow[r] & \text{Geometry} \arrow[r] & \text{Rasterizer}\end{tikzcd}$$

The GEOMETRY stage


Infinitely extending viewing frustum formed from viewer's eye through the corners of the display screen window
Polygon in world
Display screen window showing polygon's projection
Viewer's eye

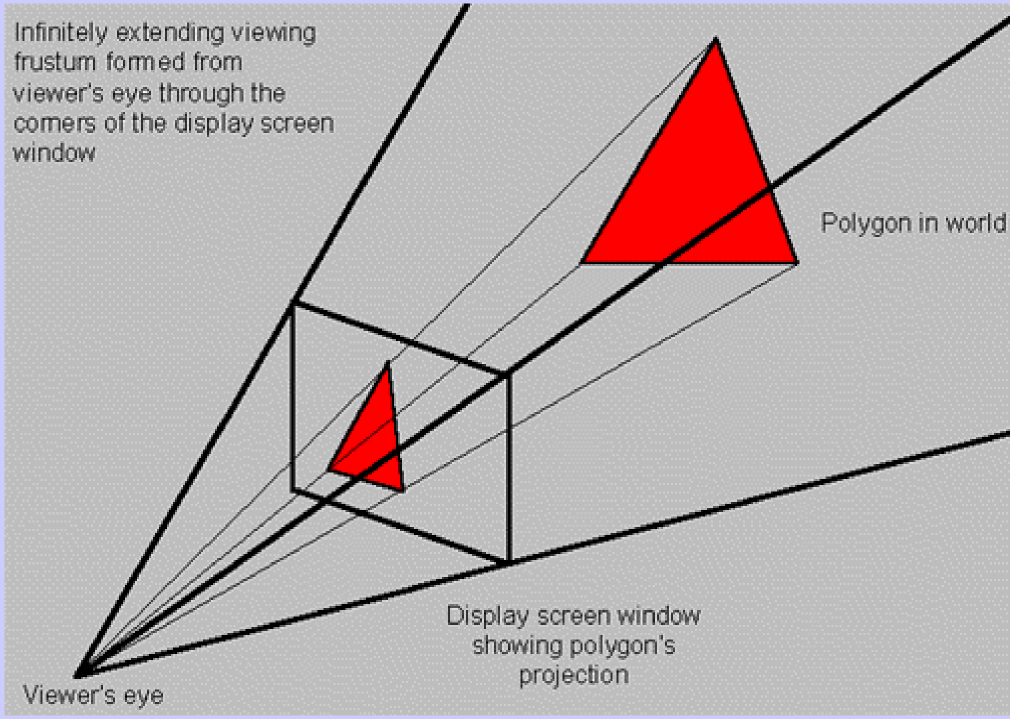
- Task: ”geometrical” operations on the input data (e.g. triangles)
- Allows:
  - Move objects (matrix multiplication)
  - Move the camera (matrix multiplication)
  - Lighting computations per triangle vertex
  - Project onto screen (3D to 2D)
  - Clipping (avoid triangles outside screen)
  - Map to window

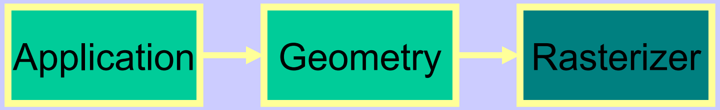
Application → Geometry → Rasterizer

The GEOMETRY stage

Model & View Transform → Vertex Shading → Projection → Clipping → Screen Mapping

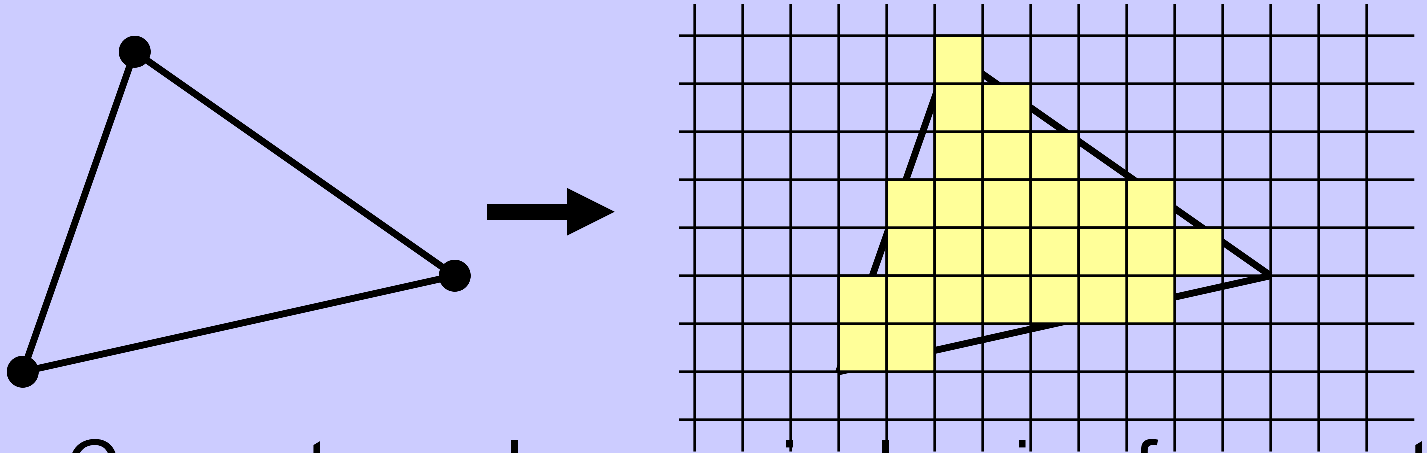
- (Instances)
- Vertex Shader
  - A program executed per vertex
    - Transformations
    - Projection
    - E.g., color per vertex
- Clipping
- Screen Mapping


Infinitely extending viewing frustum formed from viewer's eye through the corners of the display screen window
Polygon in world
Display screen window showing polygon's projection
Viewer's eye



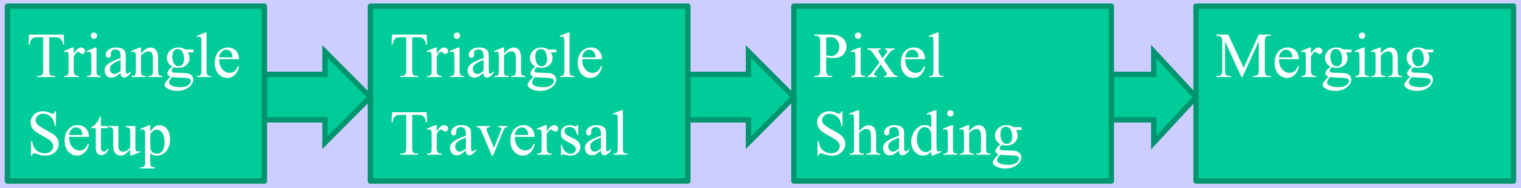
The RASTERIZER stage

- Main task: take output from GEOMETRY and turn into visible pixels on screen



- Computes color per pixel, using fragment shader (=pixel shader)
  - textures, (light sources, normal), colors and various other per-pixel operations
- And visibility is resolved here: sorts the primitives in the z-direction

The rasterizer stage



Triangle Setup:

- collect three vertices + vertex shader output (incl. normals) and make one triangle.

Triangle Traversal

- Scan conversion

Pixel Shading

- Compute pixel color

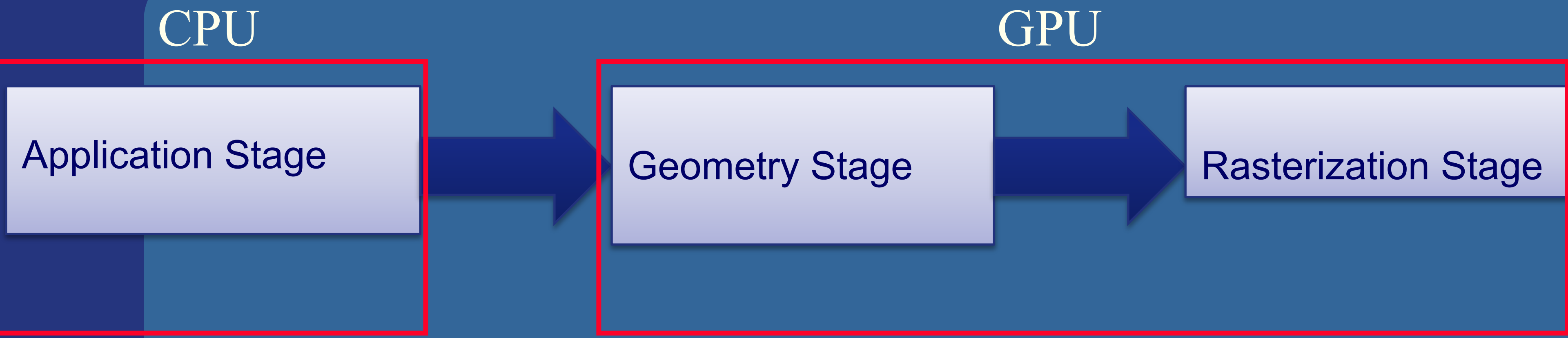
Merging:

- output color to screen

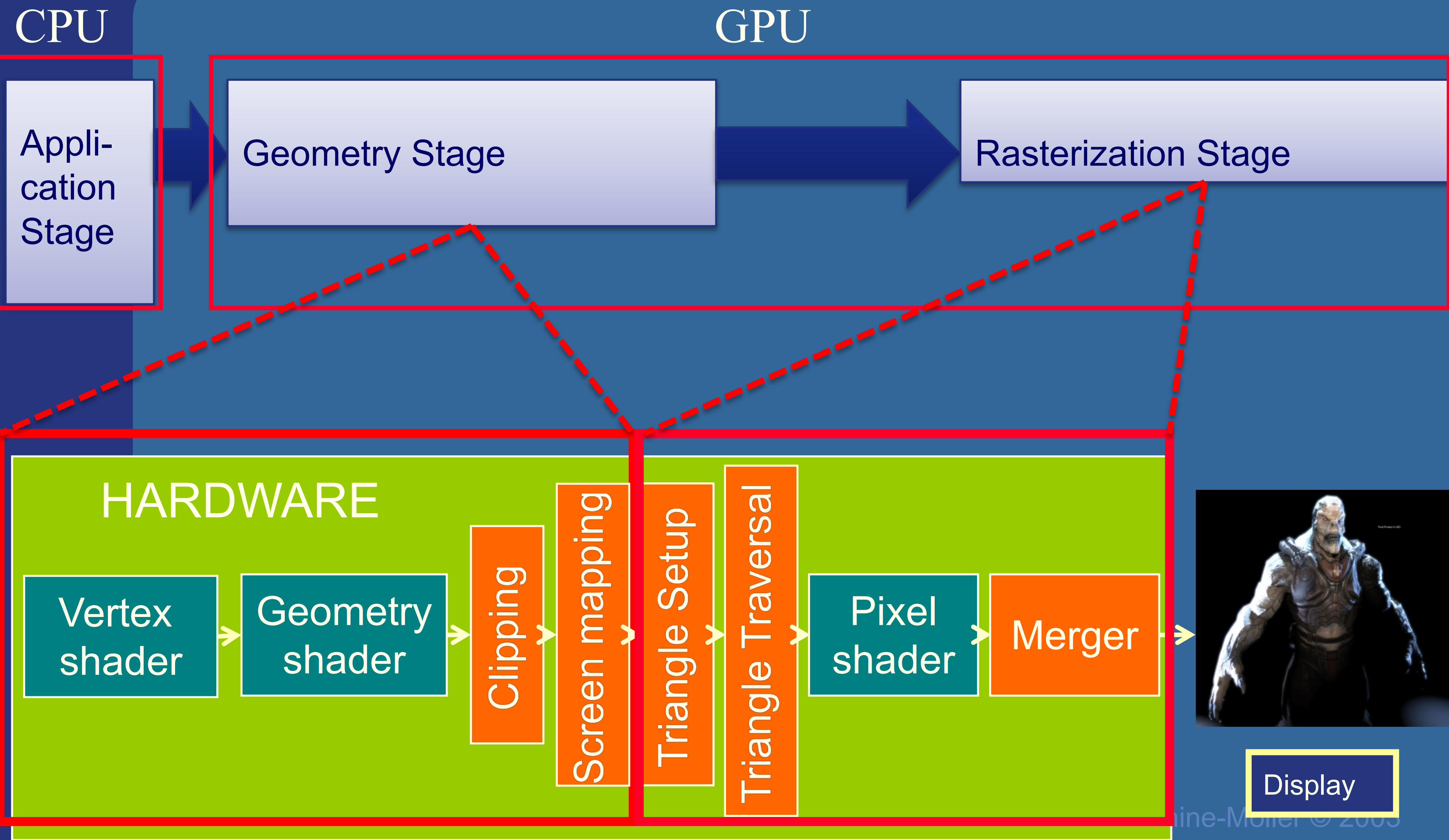
The three stages' correlation to hardware

The Application stage, geometry stage, and rasterizer stage

Rendering Pipeline and Hardware



Rendering Pipeline and Hardware

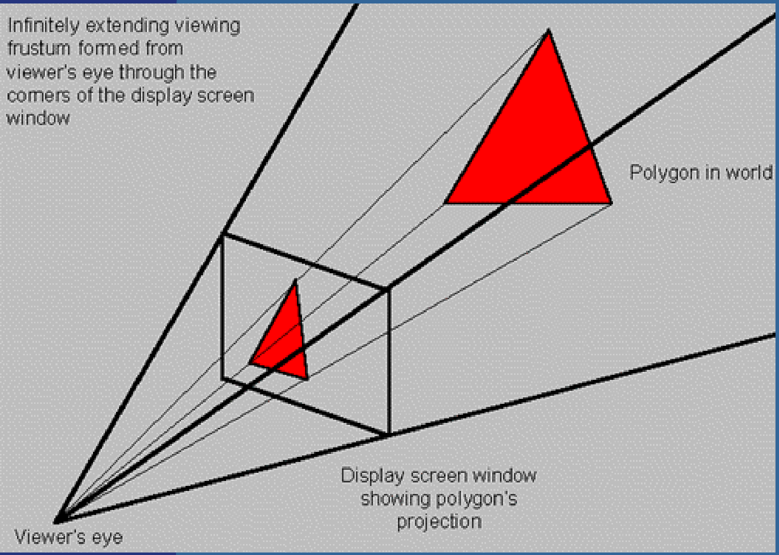


Hardware design

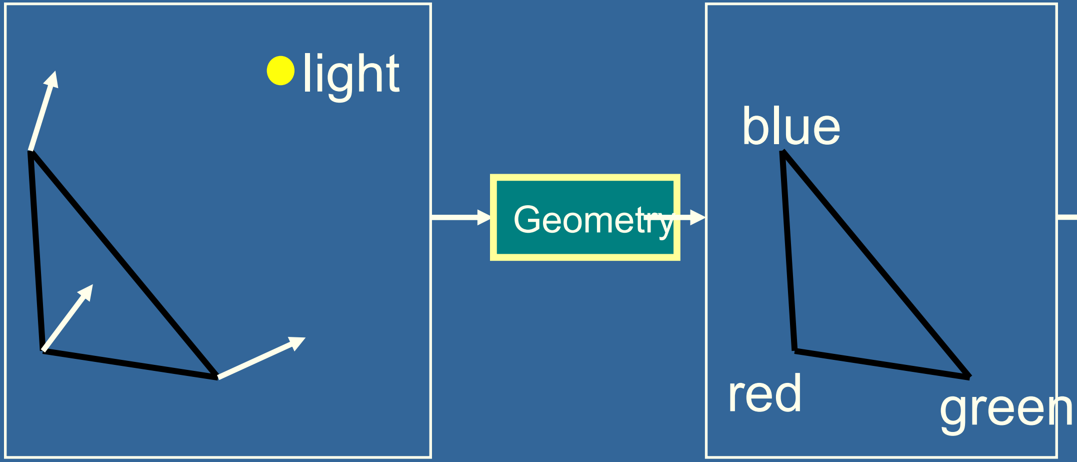
Geometry Stage

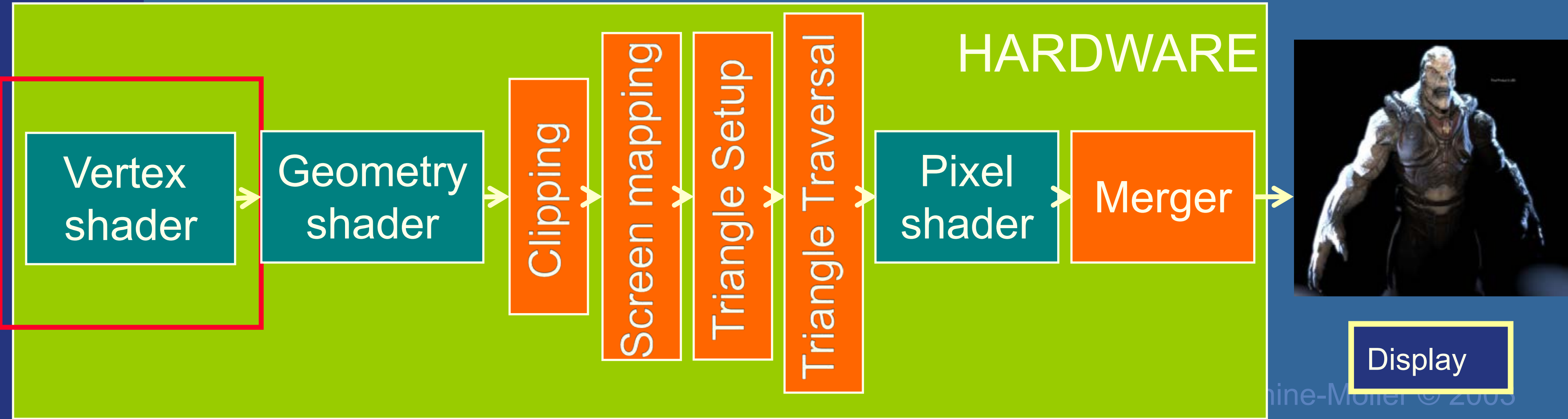
Vertex shader:

- Lighting (colors)
- Screen space positions



Infinitely extending viewing frustum formed from viewer's eye through the corners of the display screen window
Polygon in world
Display screen window showing polygon's projection
Viewer's eye



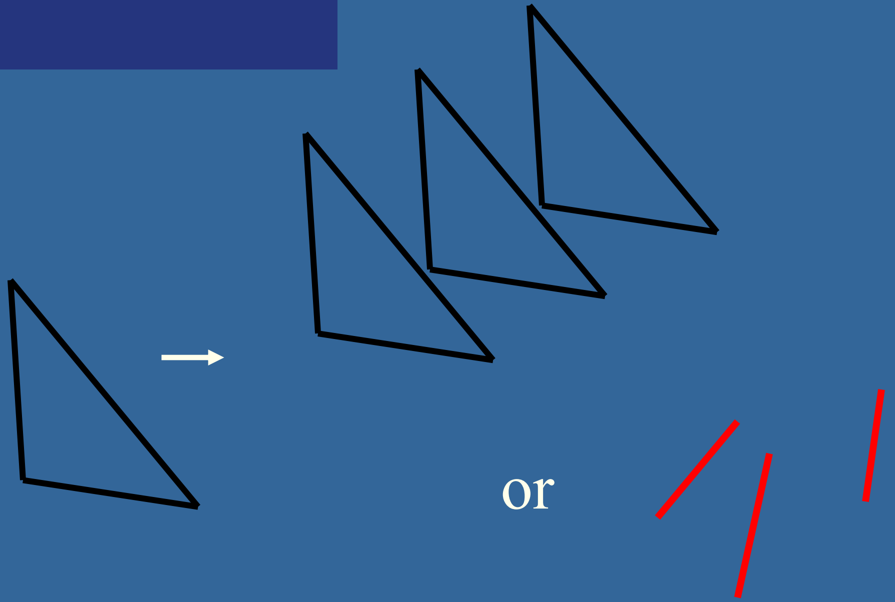


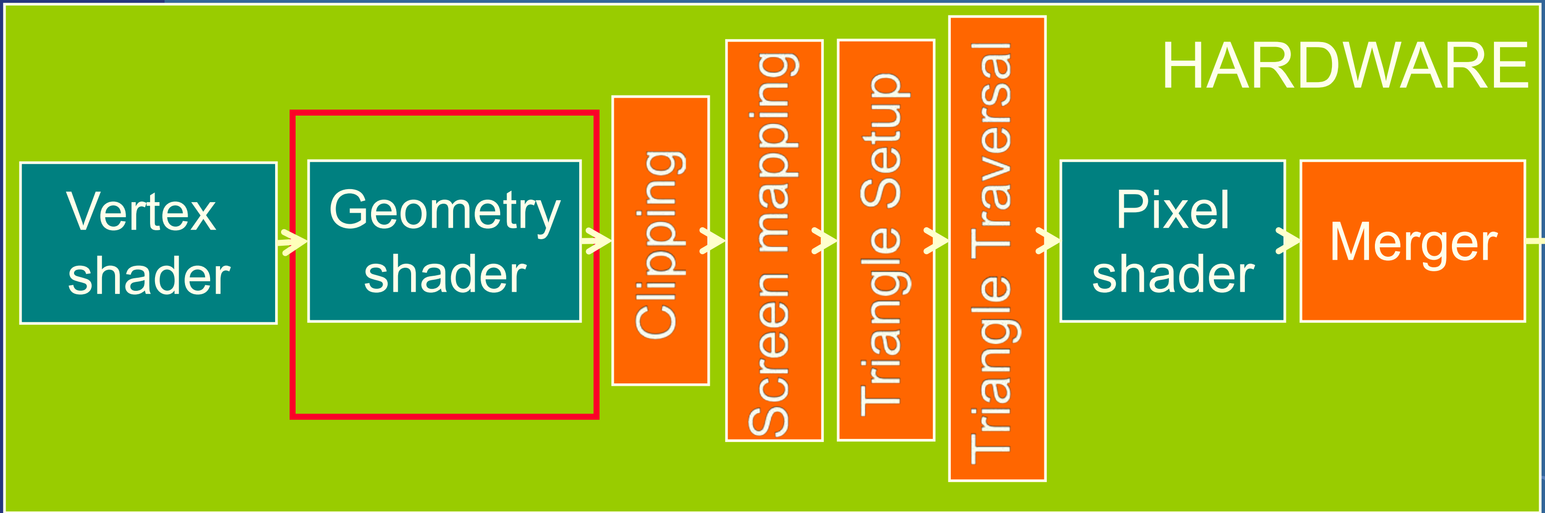
Hardware design

Geometry Stage

Geometry shader:

- One input primitive
- Many output primitives



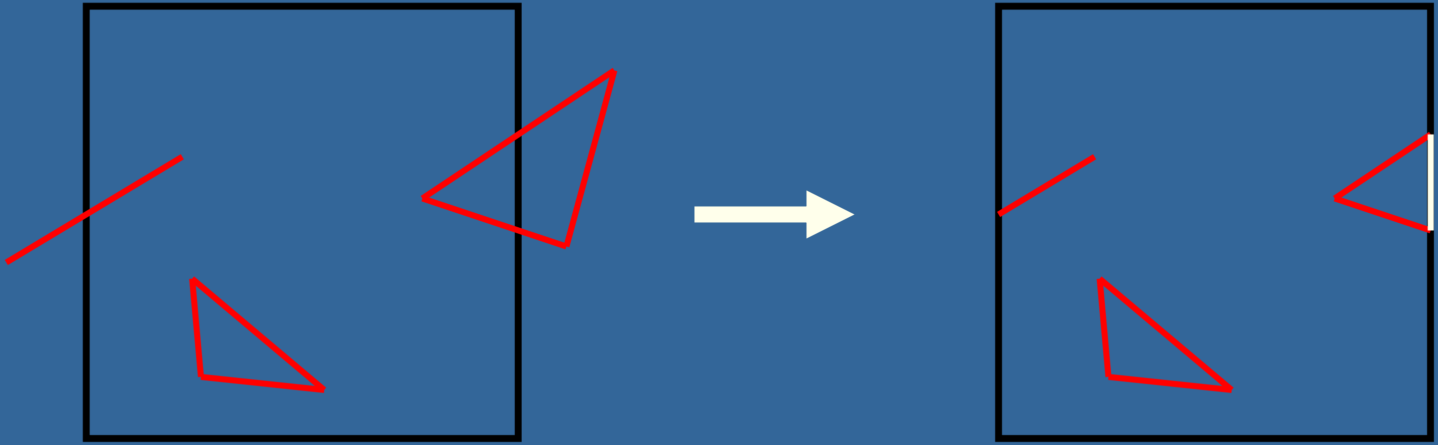


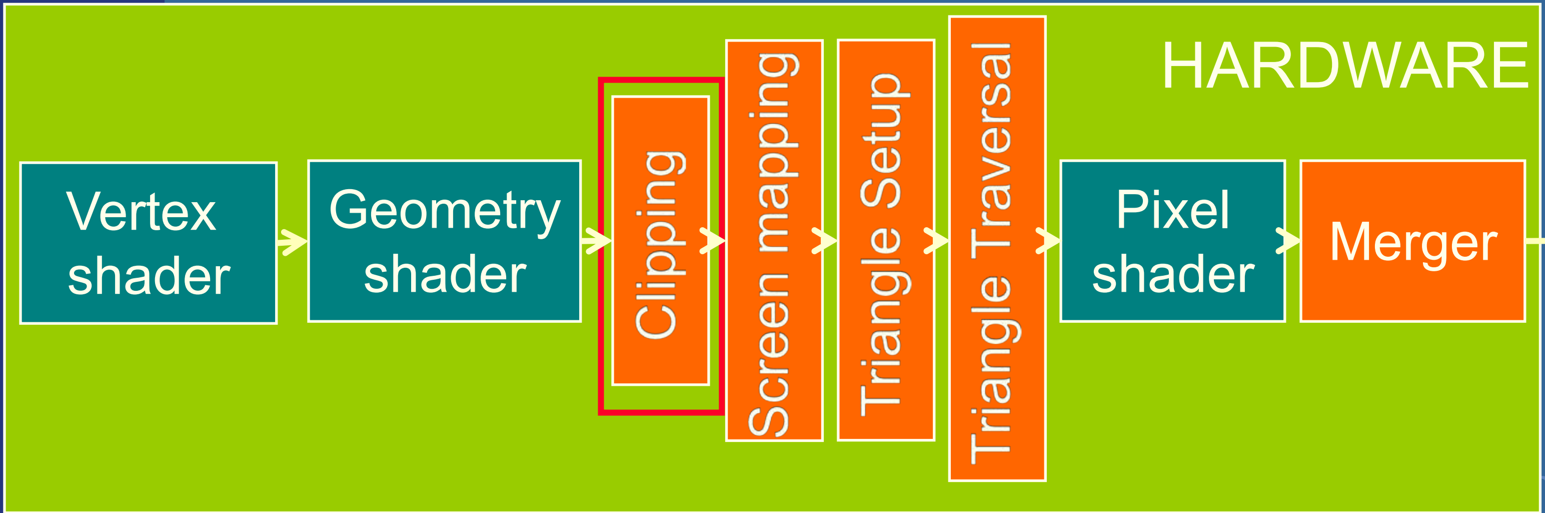
Display

Hardware design

Geometry Stage

Clips triangles against the unit cube (i.e., ”screen borders”)





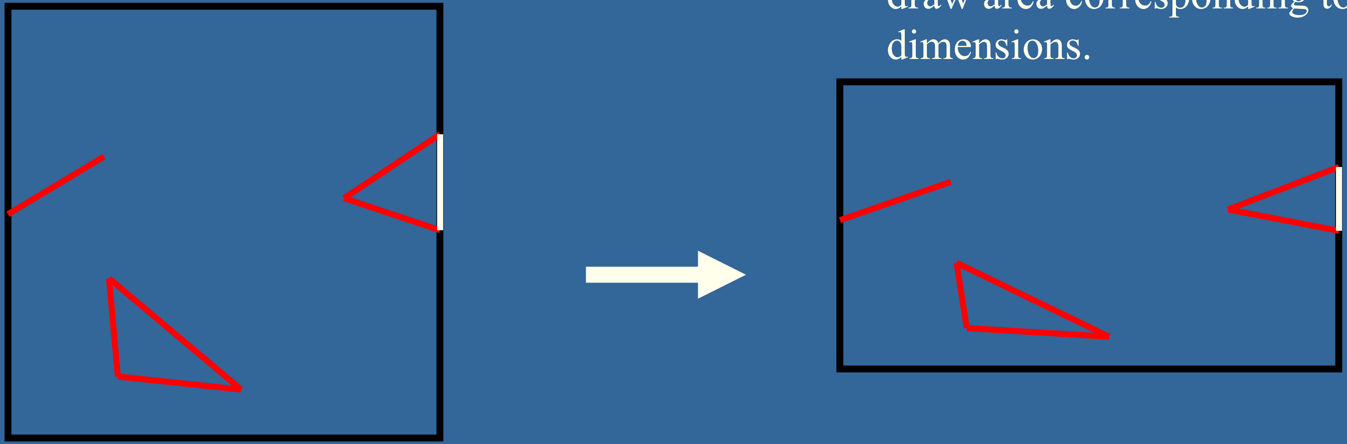
Display

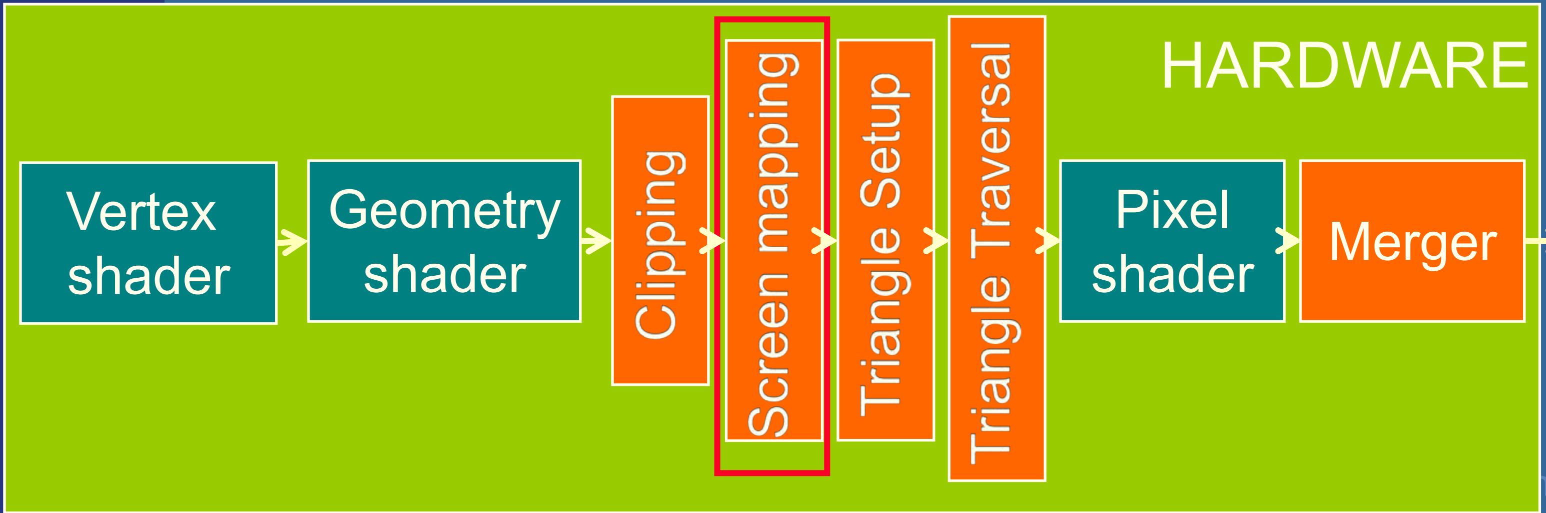
Hardware design

Rasterizer Stage

Maps window size to unit cube

Geometry stage always operates inside a unit cube [-1,-1,-1]-[1,1,1]
Next, the rasterization is made against a draw area corresponding to window dimensions.



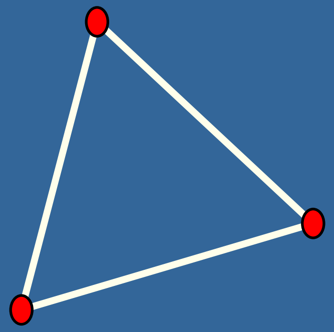


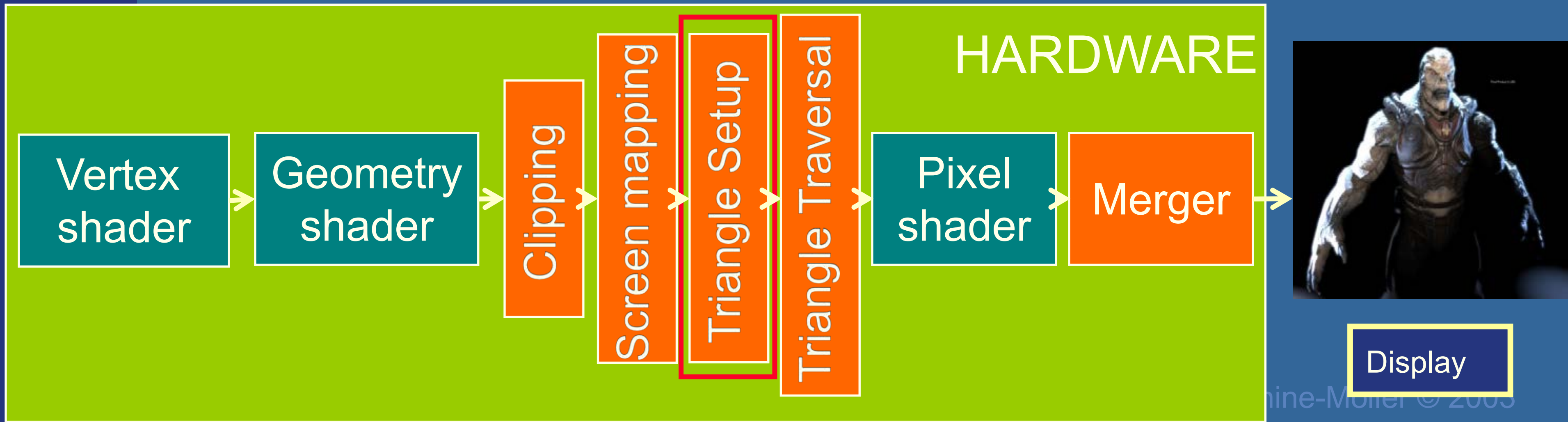
Display

Hardware design

Rasterizer Stage

Collects three vertices into one triangle

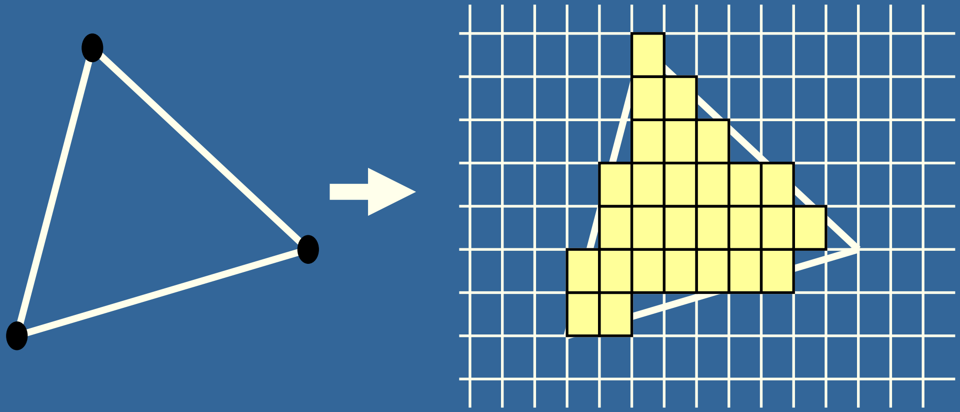


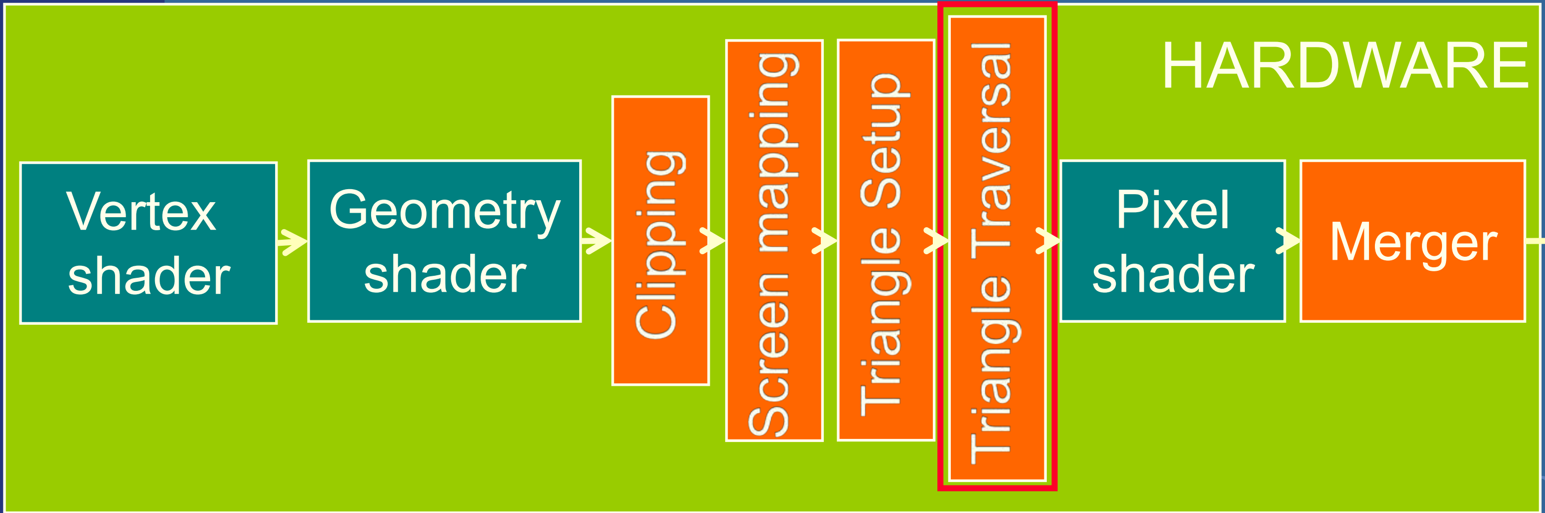


Hardware design

Rasterizer Stage

Creates the fragments/pixels for the triangle

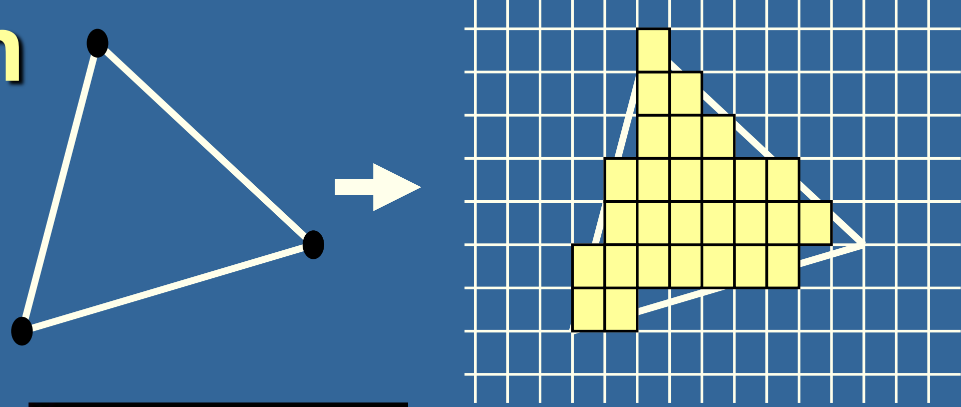


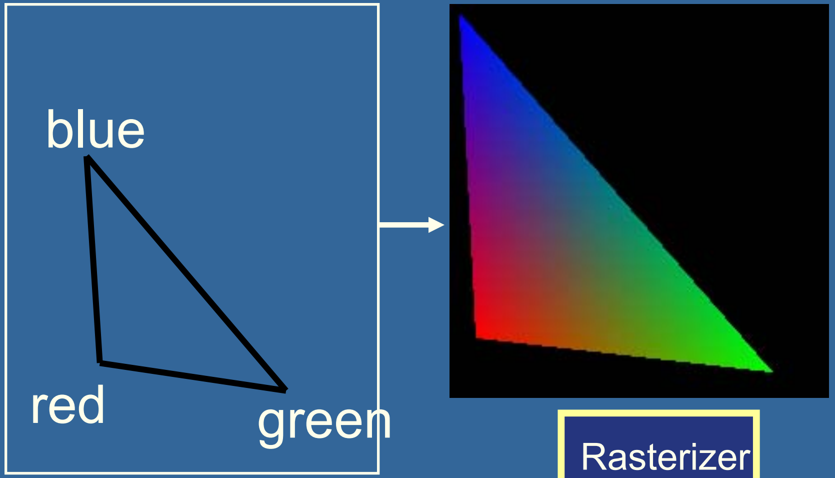


Display

Hardware design

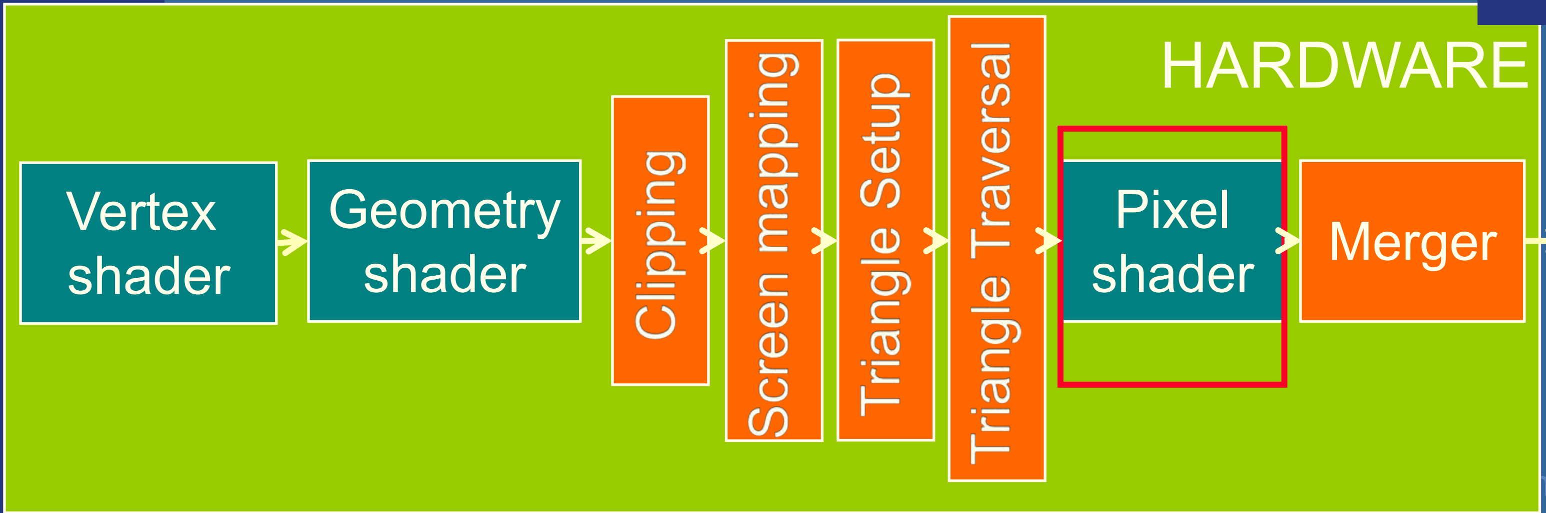
Rasterizer Stage





Rasterizer

Pixel Shader:
Compute color using:
•Textures
•Interpolated data (e.g. Colors + normals) from vertex shader

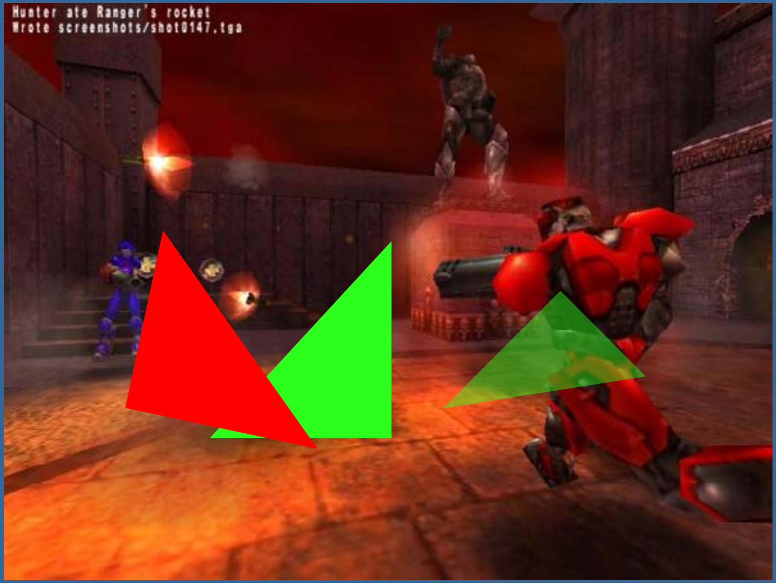


Display

Hardware design

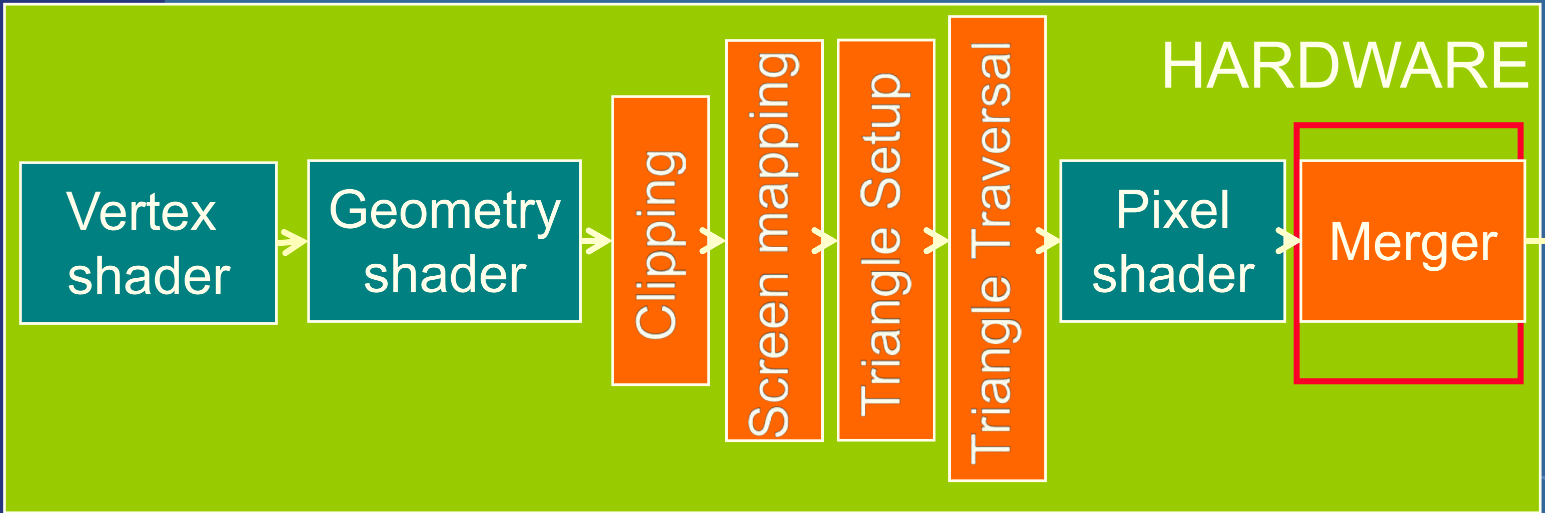
Rasterizer Stage

The merge units update the frame buffer with the pixel’s color


Hunter ate Ranger's rocket
Wrote screenshots/shot0147.tga

Frame buffer:

- Color buffers
- Depth buffer
- Stencil buffer

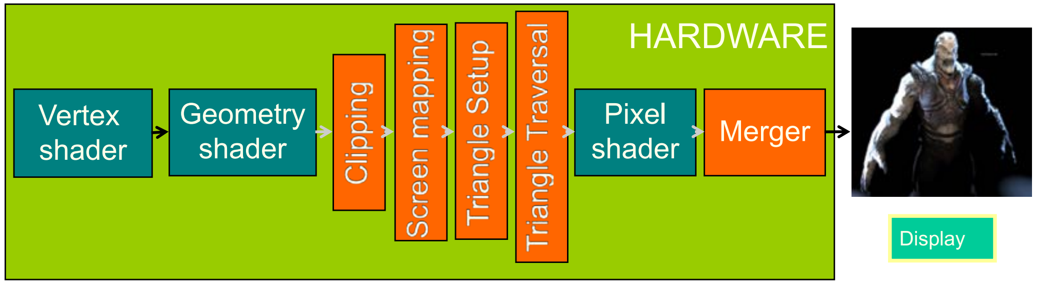


Display

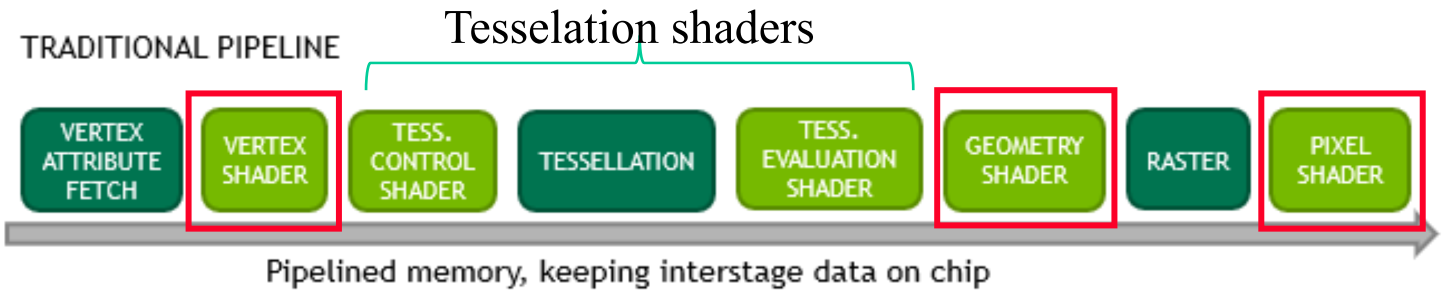
Graphics Pipeline

BONUS

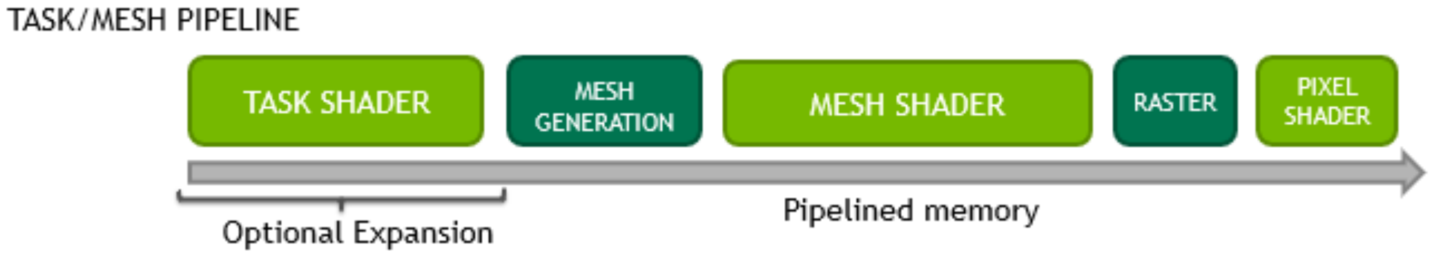
We focus on:



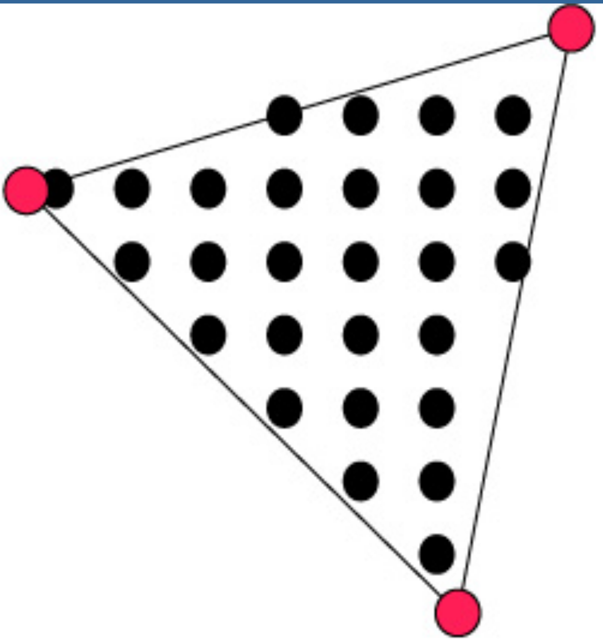
Full pipeline of today:



Next step by NVIDIA: Mesh shaders



What is vertex and fragment (pixel) shaders?

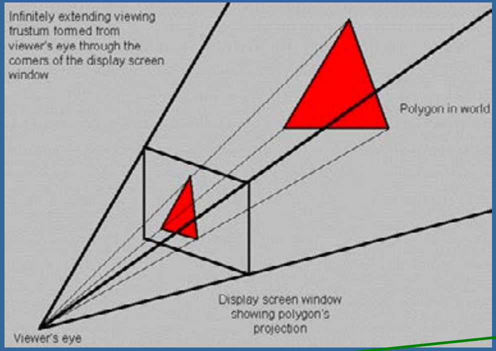


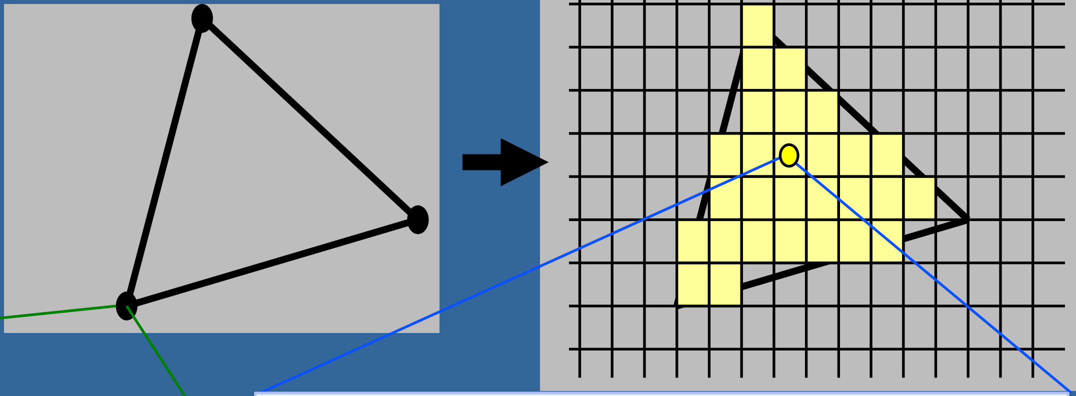
- Vertex shader: reads from textures Writes output data per vertex, which are interpolated and input to each fragment shader invocation.
- Fragment shader: reads from textures, writes to pixel color
- Memory: Texture memory (read + write) typically 1 GB – 8 GB
- Program size: the smaller the faster

🔴 For each vertex, a vertex program (vertex shader) is executed

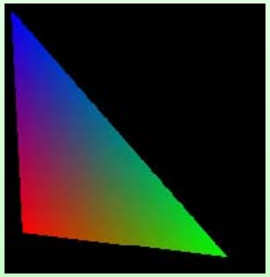
● For each fragment (pixel) a fragment program (fragment shader) is executed

Shaders


Infinitely extending viewing frustum formed from viewer's eye through the corners of the display screen window
Polygon in world
Display screen window showing polygon's projection
Viewer's eye



```
// Vertex Shader
#version 420

layout(location = 0) in vec3 vertex;
layout(location = 1) in vec3 color;
out vec3 outColor;
uniform mat4 modelViewProjectionMatrix;

void main()
{
        gl_Position = modelViewProjectionMatrix *
                vec4(vertex,1);
        outColor = color;
}
```

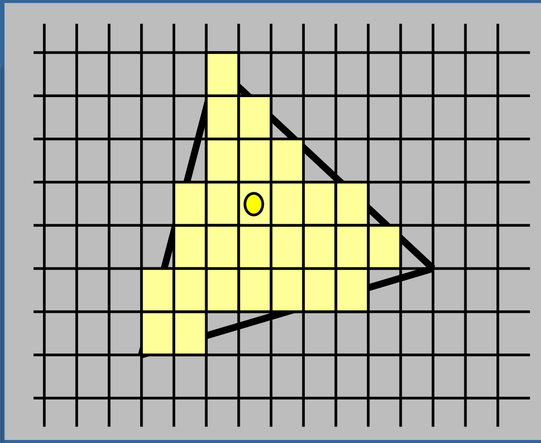


```
// Fragment Shader:
#version 420
precision highp float;

in  vec3 outColor;
layout(location = 0) out vec4 fragColor;
// Here, location=0 means that we draw to
      frameBuffer[0], i.e., the screen.

void main()
{
      fragColor = vec4(outColor,1);
}
```

Shaders

```
precision highp float;

uniform sampler2D tex0;
uniform sampler2D tex1;
uniform sampler2D tex2;
uniform sampler2D tex3;

uniform float val;

varying vec2 uv_0;
varying vec3 n;

void main(void) {
        gl_FragColor.rgb = compute_color();
        gl_FragColor.a = 1.0;
}
```

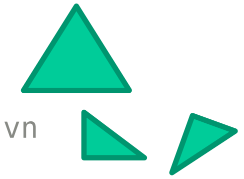
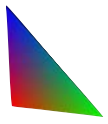
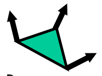
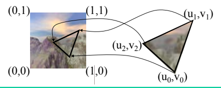
```
vec3 compute_color()
{
   vec4 gbuffer = texture2D(tex0, uv_0);
        int intColor = int(gbuffer.x);
        int r = (intColor/256)/256;
        intColor -= r*256*256;
        int g = intColor/256;
        intColor -= g*256;
        int b = intColor;
         vec3 color = vec3(float(r)/255.0, float(g)/255.0,
        float(b)/255.0 );

        normal = vec3(sin(gbuffer.g) * cos(gbuffer.b),
        sin(gbuffer.g)*sin(gbuffer.b), cos(gbuffer.g));
        vec2 ang = gbuffer.gb*2.0-vec2(1.0);
        vec2 scth = vec2( sin(ang.x * PI), cos(ang.x * PI);
        vec2 scphi = vec2(sqrt(1.0 - ang.y*ang.y), ang.y);
        normal = -vec3(scth.y*scphi.x, scth.x*scphi.x, scphi.y);
        roughness = 0.05;
        specularity = 1.0;
        fresnelR0 = 0.3;
        return color;
}
```



OpenGL

CPU-side
Language: C++
API: OpenGL (Direct3D)
Window system: SDL (Cocoa, Win32,…)

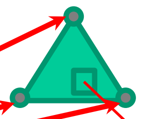
C++:
```
float positions[] = {
//   X       Y     Z   per vertex
   0.0f,   0.5f, 1.0f, // v0
  -0.5f,  -0.5f, 1.0f, // v1
   0.5f,  -0.5f, 1.0f  // v2
   ...                 // ... vn
};
// and any other per-vertex data, e.g.:
float colors[] = {
//   R       G     B
   1.0f,   0.0f, 0.0f, // c0
  -0.0f,   1.0f, 0.0f, // c1
   0.0f,   0.0f, 1.0f  // c2
   ...                 // ... cn
};
float normals[] = {...};
float textureCoords[] = {...};
```

OpenGL:
```
 Vertex-buffer objects
uint32 positionBuffer; // x,y,z per vertex
uint32 colorBuffer;    // r,g,b per vertex

 Vertex-Array object // groups the arrays
uint32 vertexArrayObject;

 Shaders
uint32 vertexShader;
uint32 fragmentShader;
uint32 shaderProgram;
```

GPU-side
Language: GLSL
used for vertex- ,geometry-, and fragment shaders

Vertex Shader:
```
#version 420

layout(location = 0) in  vec3 position;
layout(location = 1) in  vec3 color;

out vec3 outColor; // r,g,b

void main()
{
   gl_Position = vec4(position, 1.0);
   outColor = color;
}
```



Fragment Shader:
```
#version 420

precision highp float; // required by GLSL spec Sect 4.5.3
                       // (though nvidia does not, amd does)

layout(location = 0) out vec4 fragmentColor;
in vec3 outColor;

void main()
{
   // fragmentColor = vec4(1,1,1,1);
   fragmentColor.rgb = outColor;
   fragmentColor.a = 1.0;
}
```

Per-pixel-interpolated value

CPU-side

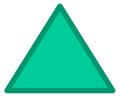
Language: C++

API: OpenGL (Direct3D)

Window system: SDL (Cocoa, Win32,…)

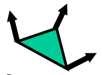
C++:

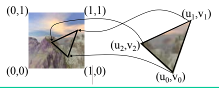
```
float positions[] = {
//   X       Y      Z
   0.0f,   0.5f, 1.0f, // v0
  -0.5f,  -0.5f, 1.0f, // v1
   0.5f,  -0.5f, 1.0f  // v2
```



How to connect the vertexArrayObject as vertex shader input (position, color):

```
   0.0f,   0.0f, 1.0f  // c2
   ...                // ... cn
};
float normals[] = {...};
float textureCoords[] = {...};
```





```
OpenGL:
 Vertex-buffer objects
uint32 positionBuffer; // x,y,z per vertex
uint32 colorBuffer;    // r,g,b per vertex

 Vertex-Array object // groups the arrays
uint32 vertexArrayObject;

 Shaders
uint32 vertexShader;
uint32 fragmentShader;
uint32 shaderProgram;
```

GPU-side

Language: GLSL

used for vertex- ,geometry-, and fragment shaders

Vertex Shader:

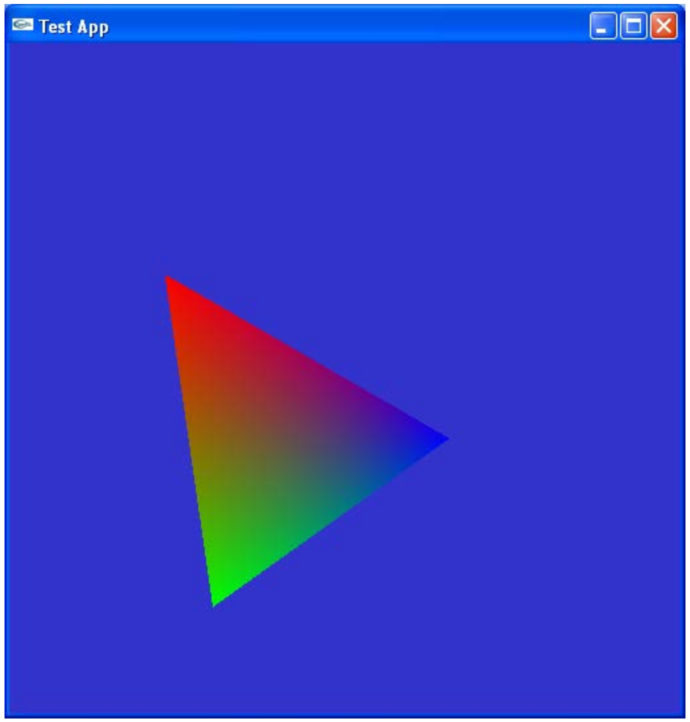
```
#version 420

layout(location = 0) in  vec3 position;
layout(location = 1) in  vec3 color;

out vec3 outColor;

void main()
{
   gl_Position = vec4(position, 1.0);
   outColor = color;
}
```

```
glGenVertexArrays(1, &vertexArrayObject);
// Following commands now affect this vertex array object.
glBindVertexArray(vertexArrayObject);

// Makes positionBuffer the current array buffer for subse
quent commands.
glBindBuffer( GL_ARRAY_BUFFER, positionBuffer );
// Attach positionBuffer to vertexArrayObject,
// in location 0. 3 floats per vertex
glVertexAttribPointer(0, 3, GL_FLOAT, …);

// Makes colorBuffer the current array buffer for subseque
nt commands.
glBindBuffer( GL_ARRAY_BUFFER, colorBuffer );
// Attaches colorBuffer to vertexArrayObject,
// in location 1. 3 floats per vertex
glVertexAttribPointer(1, 3, GL_FLOAT, … );

glEnableVertexAttribArray(0); // Enable attribute array 0
glEnableVertexAttribArray(1); // Enable attribute array 1
```

A Simple Program

Computer Graphics version of "Hello World"

Generate a triangle on a solid background



Simple Application...

```
int main(int argc, char *argv[])
{
    // open window of size 512x512 with double buffering, RGB colors, and Z-buffering
    g_window = labhelper::init_window_SDL("OpenGL Lab 1”, 512, 512);
    initGL(); // Set up our shaderProgram and our vertexArrayObject
    while (true) {

        display(); // render our geometry

        SDL_GL_SwapWindow(g_window); // swap front/back buffer. Ie., displays the frame.


        SDL_Event event;
        while (SDL_PollEvent(&event)) {
            if (event.type == SDL_QUIT || (event.type == SDL_KEYUP &&
                event.key.keysym.sym == SDLK_ESCAPE)) {
                    labhelper::shutDown(g_window);
                    return 0;
            }
        }
    }
    return 0;
}
```

```
void display(void)
{
	// The viewport determines how many pixels we are rasterizing to
	int w, h;
	SDL_GetWindowSize(g_window, &w, &h);
	glViewport(0, 0, w, h);   // Set viewport

	// Clear background
	glClearColor(0.2, 0.2, 0.8, 1.0);    // Set clear color  - for background
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clears the color buffer and the z-buffer

	glDisable(GL_CULL_FACE); // Both front and back face of triangles should be visible

	// DRAW OUR TRIANGLE(S)
	glUseProgram( shaderProgram );  // Shader Program. Sets what vertex/fragment shaders to use.
	// Bind the vertex array object that contains all the vertex data.
	glBindVertexArray(vertexArrayObject);
	// Submit triangles from currently bound vertex array object.
	glDrawArrays( GL_TRIANGLES, 0, 3 );        // Render 1 triangle (i.e., 3 vertices), starting at vertex 0.

	glUseProgram( 0 );   // "unsets" the current shader program. Not really necessary.
}
```

Lab 1 will teach you this, i.e., setting up a shader program and vertex arrays.

Example of a simple GfxObject class

```
class GfxObject {
public:
         load(“filename”); // Creates m_shaderProgram + m_vertexArrayObject
         render()
         {
                  /* You may want to initiate more OpenGL states, e.g., for
                     textures (more on that in further lectures) */
                  glUseProgram(m_shaderProgram);
                  glBindVertexArray(m_vertexArrayObject);
                  glDrawArrays( GL_TRIANGLES, 0, numVertices);
         };
private:
         uint     numVertices;
         Gluint   m_shaderProgram;
         GLuint   m_vertexArrayObject;
};
```

Example:

```
GfxObject myCoolObject;
myCoolObject.load(“filename”);

In display():
         myCoolObject.render();
```

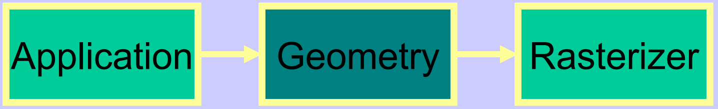
The Geometry stage and Rasterizer stage in more detail

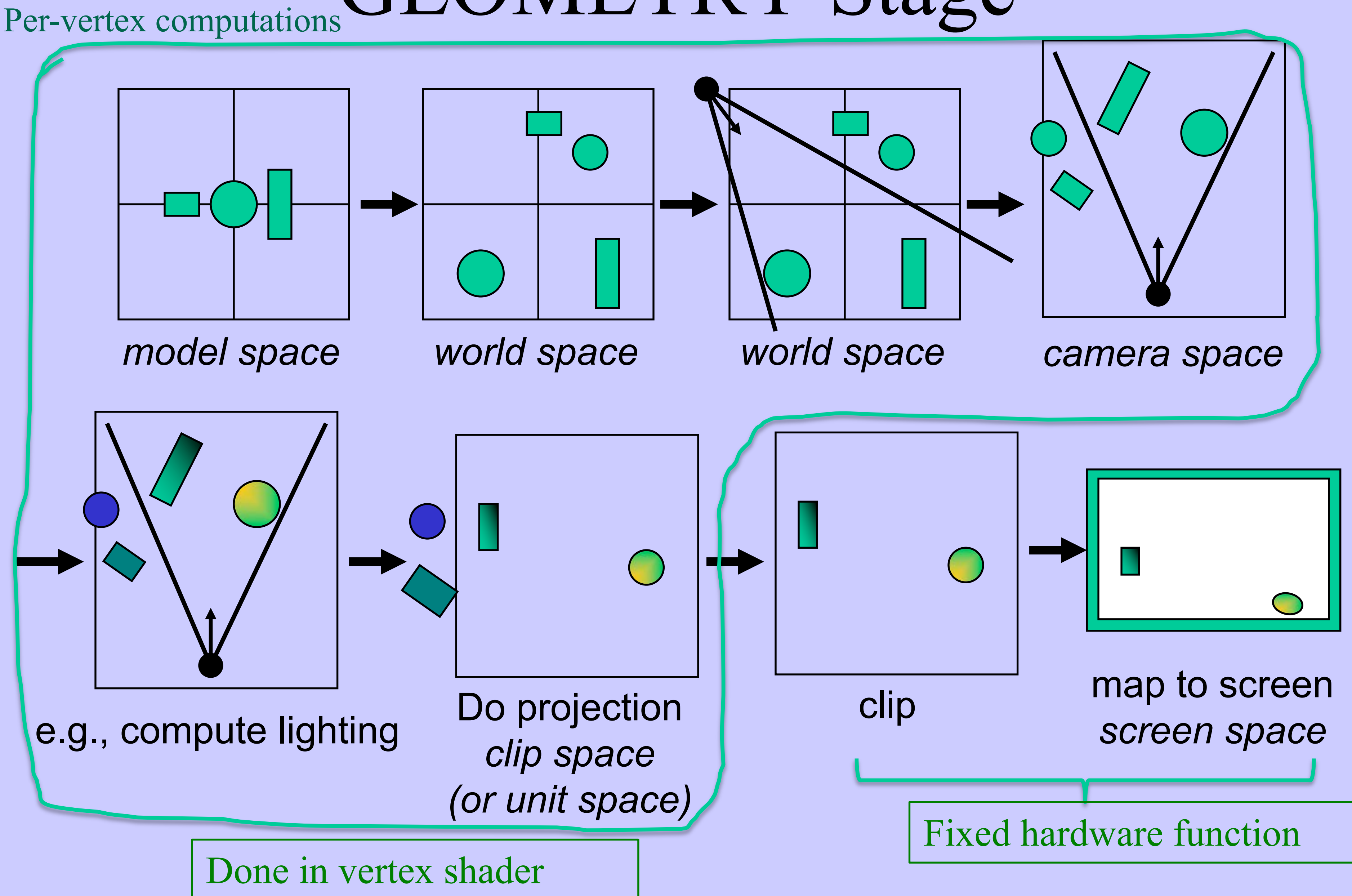
Application
Geometry
Rasterizer

Rewind! Let’s take a closer look

- The programmer ”sends” down primtives to be rendered through the pipeline (using API calls)
- The geometry stage does per-vertex operations
- The rasterizer stage does per-pixel operations
- Next, scrutinize geometry and rasterizer

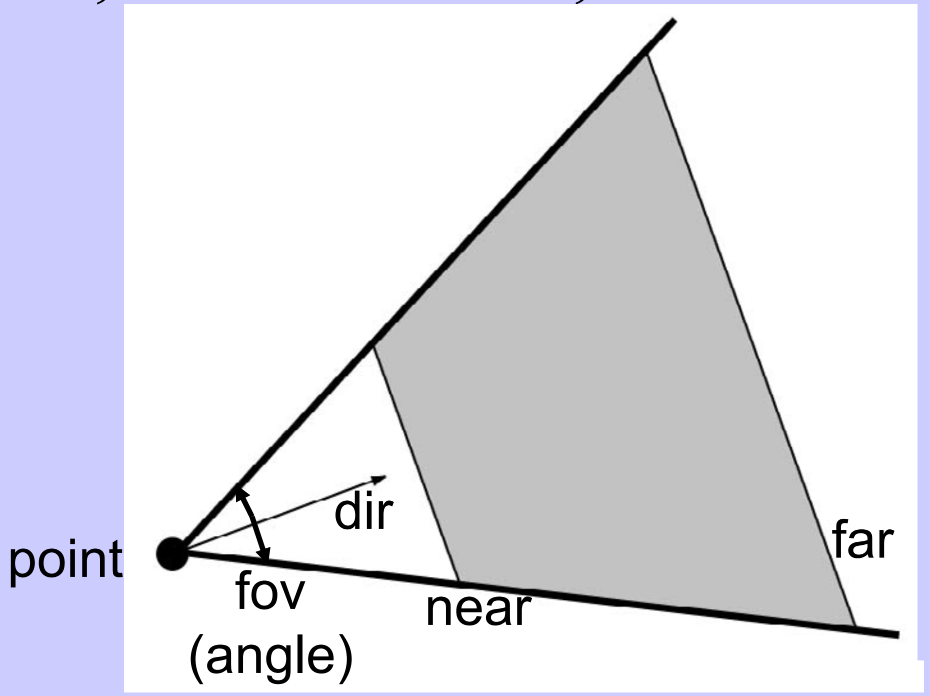
GEOMETRY Stage





Virtual Camera

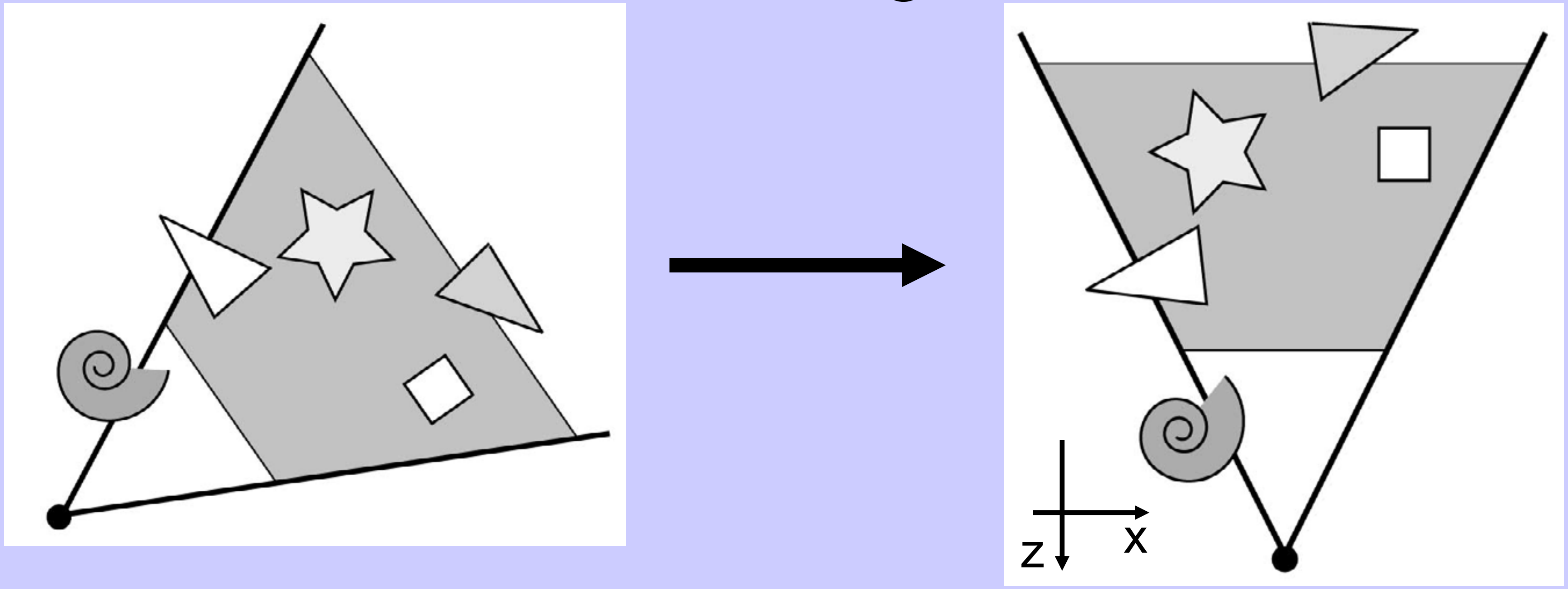
- Defined by position, direction vector, up vector, field of view, near and far plane.



- Create image of geometry inside gray region
- Used by OpenGL, DirectX, ray tracing, etc.

GEOMETRY - The view transform

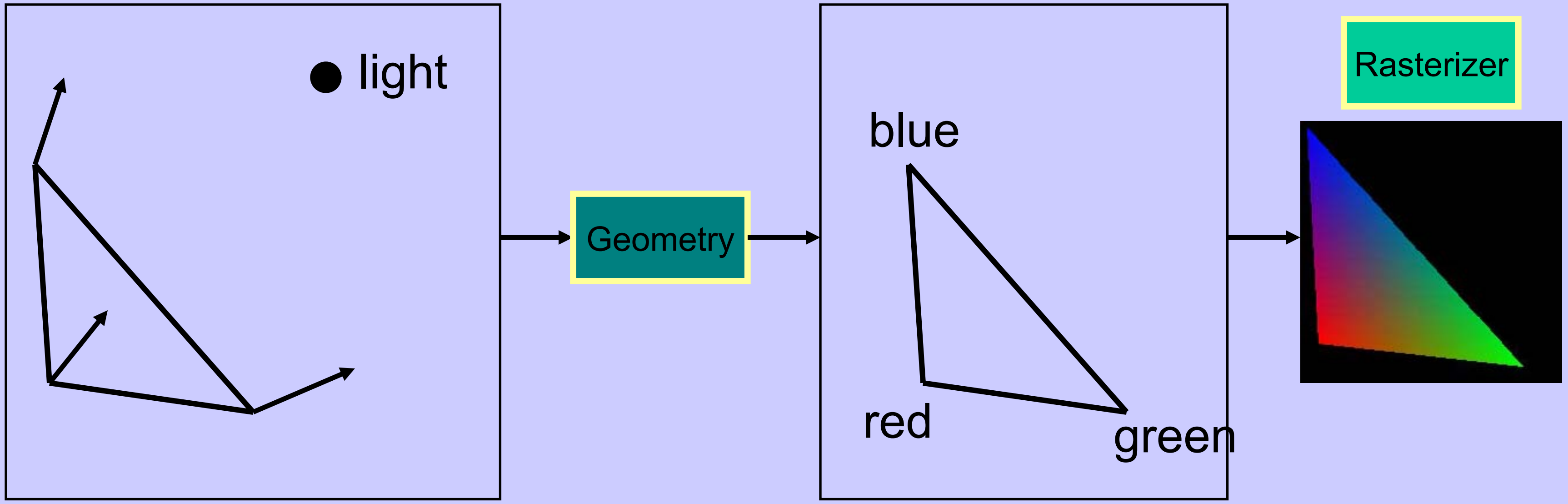
- You can move the camera in the same manner as objects
- But apply inverse transform to objects, so that camera looks down negative z-axis



Application
Geometry
Rasterizer

GEOMETRY - Lighting

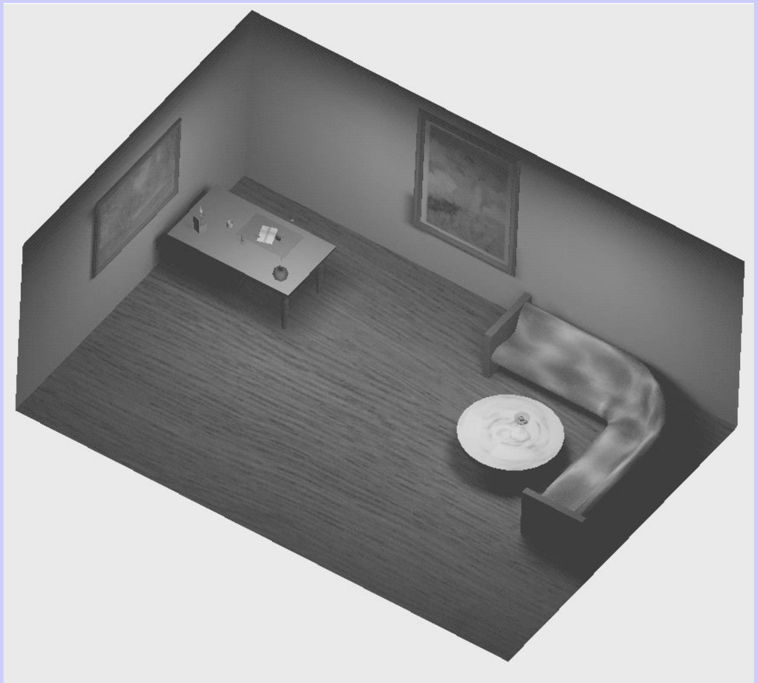
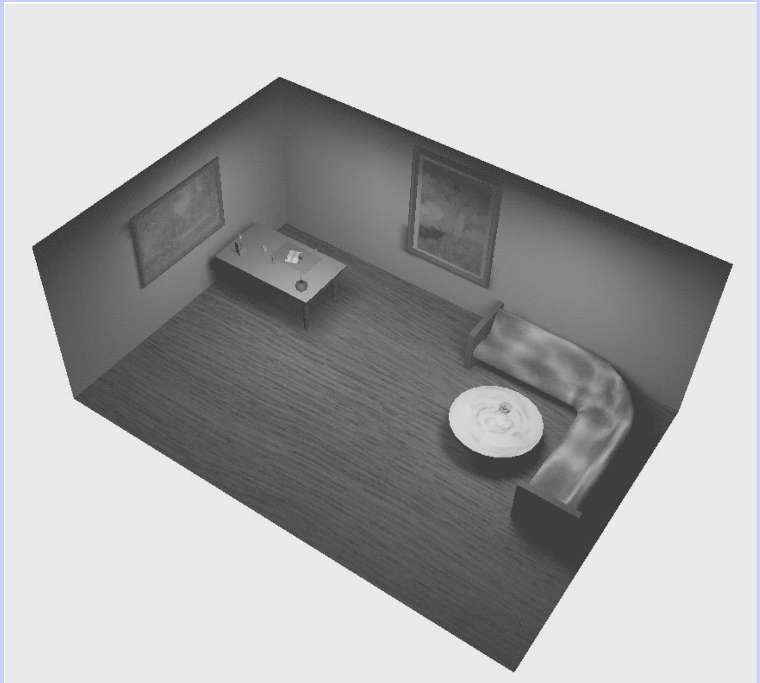
- Compute ”lighting” at vertices



- Try to mimic how light in nature behaves
  - Hard so uses empirical models, hacks, and some real theory
- Much more about this in later lecture

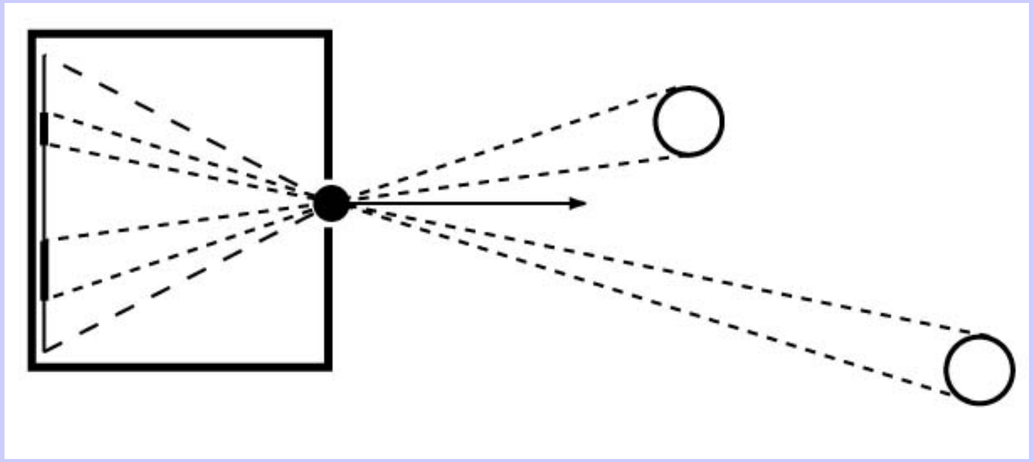
GEOMETRY - Projection

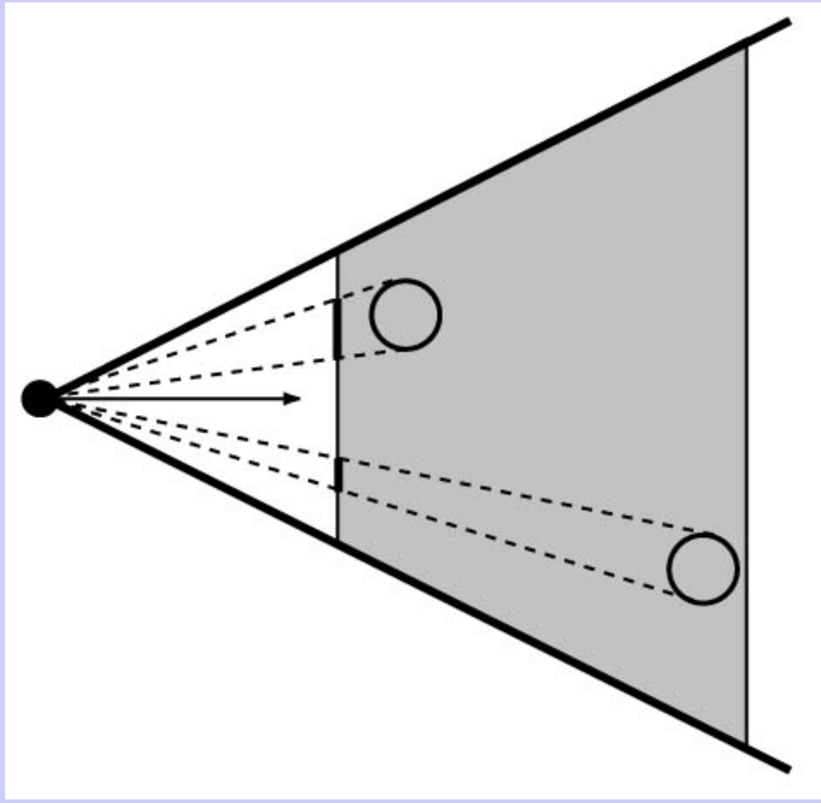
- Two major ways to do it
  - Orthogonal (useful in few applications)
  - Perspective (most often used)
    - Mimics how humans perceive the world, i.e., objects' apparent size decreases with distance

GEOMETRY - Projection

- Also done with a matrix multiplication!
- Pinhole camera (left), analog used in CG (right)



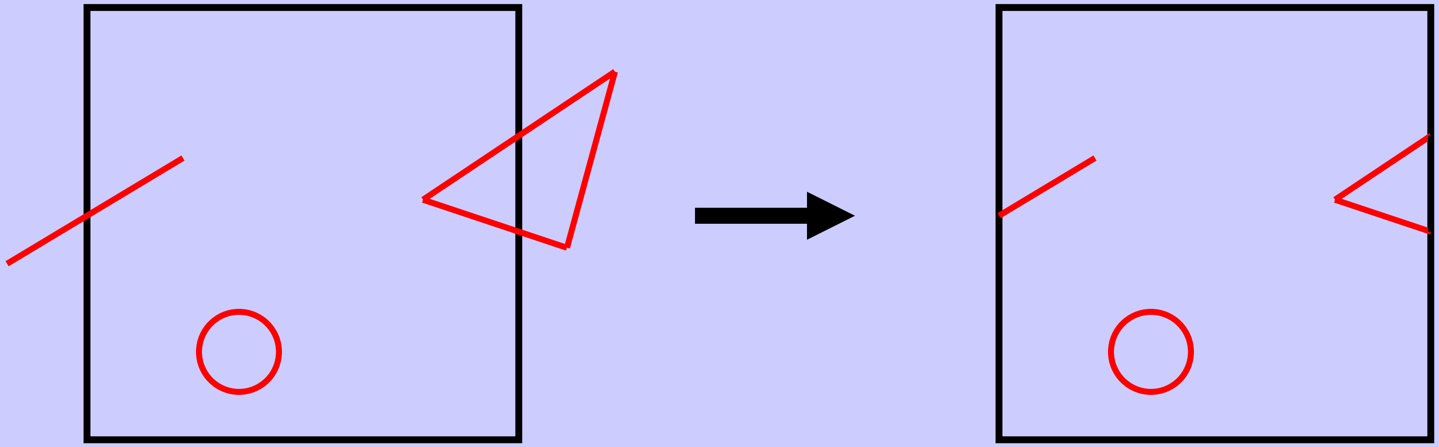


GEOMETRY

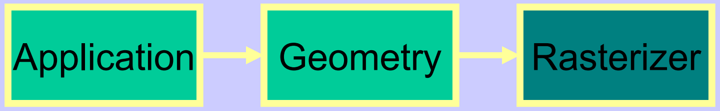
Application
Geometry
Rasterizer

Clipping and Screen Mapping

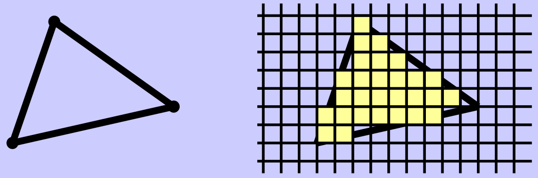
- Square (cube) after projection
- Clip primitives to square



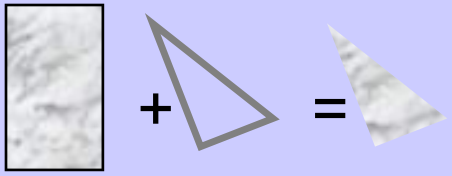
- Screen mapping, scales and translates the square so that it ends up in a rendering window
- These ”screen space coordinates” together with Z (depth) are sent to the rasterizer stage

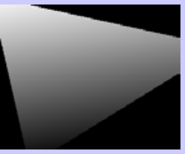


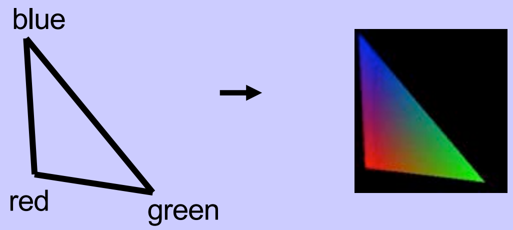
The RASTERIZER in more detail



- Scan-conversion
  - Find out which pixels are inside the primitive
- Fragment shaders
  - E.g. put textures on triangles
  - Use interpolated data over triangle
  - and/or compute per-pixel lighting







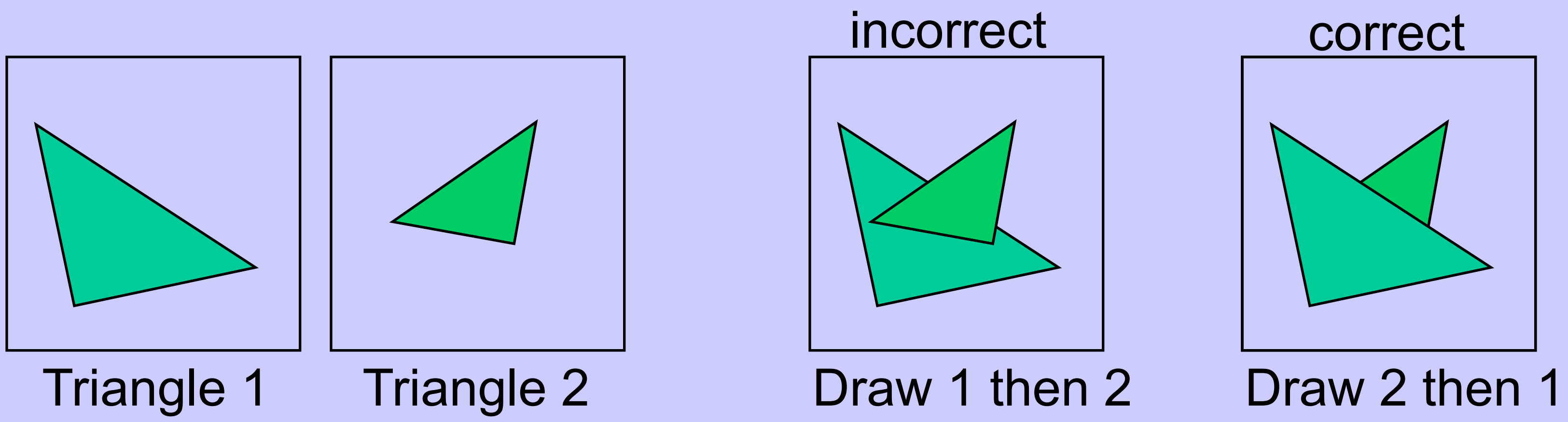
- Z-buffering
  - Make sure that what is visible from the camera really is displayed
- Doublebuffering

The RASTERIZER

Application
Geometry
Rasterizer

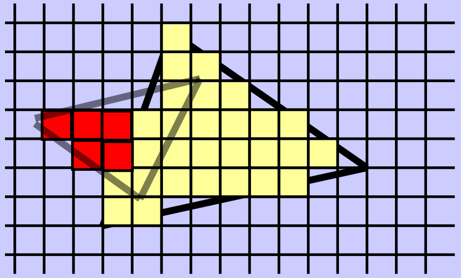
Z-buffering

- A triangle that is covered by a more closely located triangle should not be visible
- Assume two equally large tris at different depths



The RASTERIZER Z-buffering

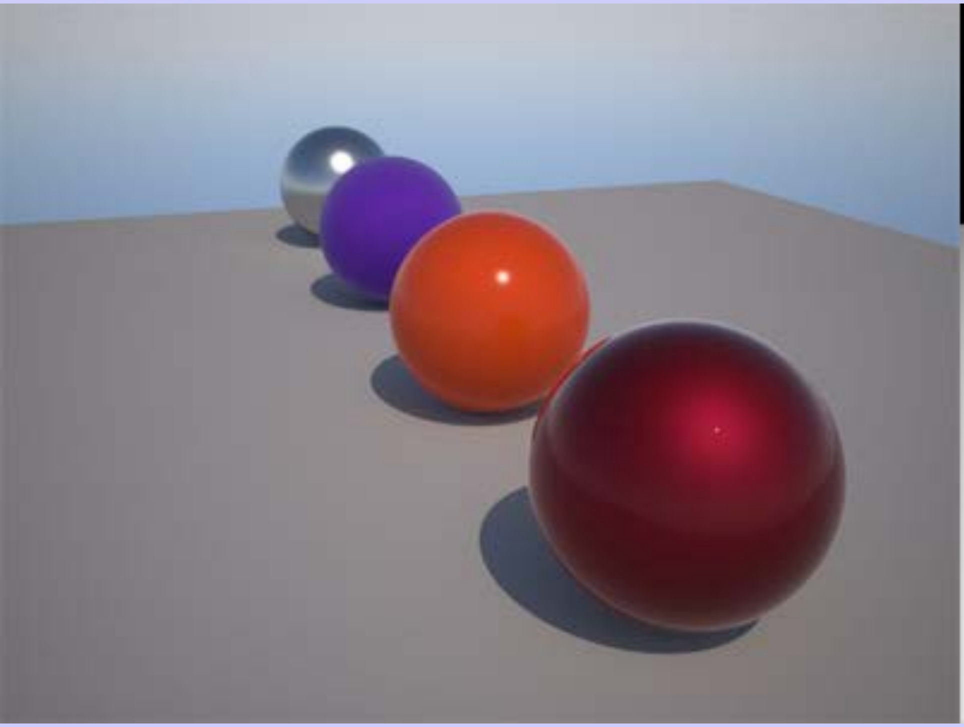
$$\begin{tikzcd}\text{Application} \arrow[r] & \text{Geometry} \arrow[r] & \text{Rasterizer}\end{tikzcd}$$



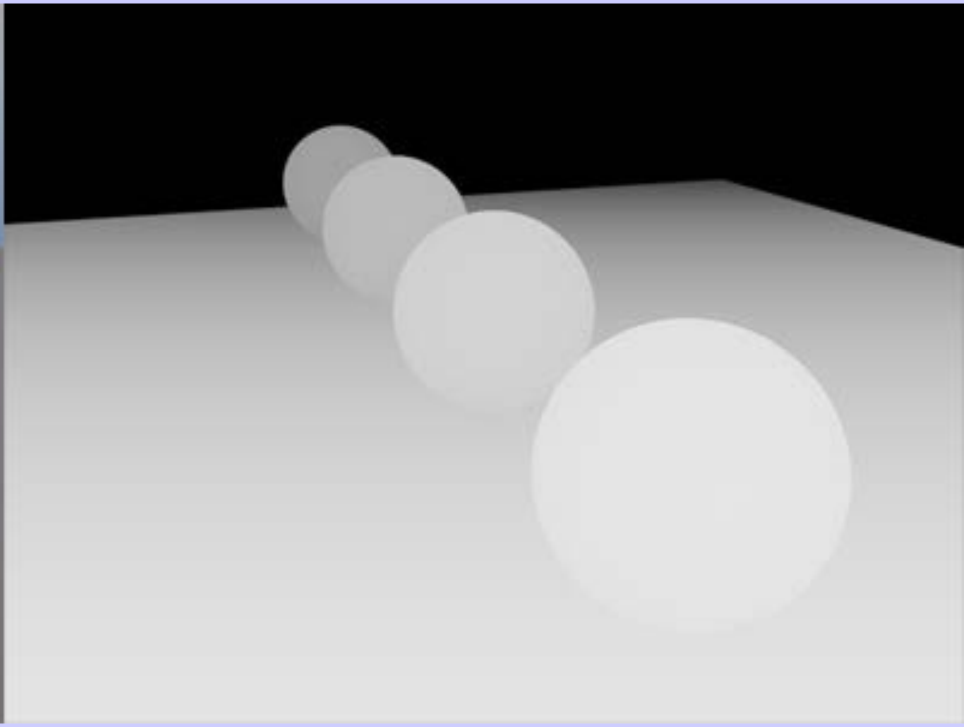
- Would be nice to avoid sorting…
- The Z-buffer (aka depth buffer) solves this
- Idea:
  - Store z (depth) at each pixel
  - When rasterizing a triangle, compute z at each pixel on triangle
  - Compare triangle's z to Z-buffer z-value
  - If triangle's z is smaller, then replace Z-buffer and color buffer
  - Else do nothing
- Can render in any order

The RASTERIZER

Z-buffer



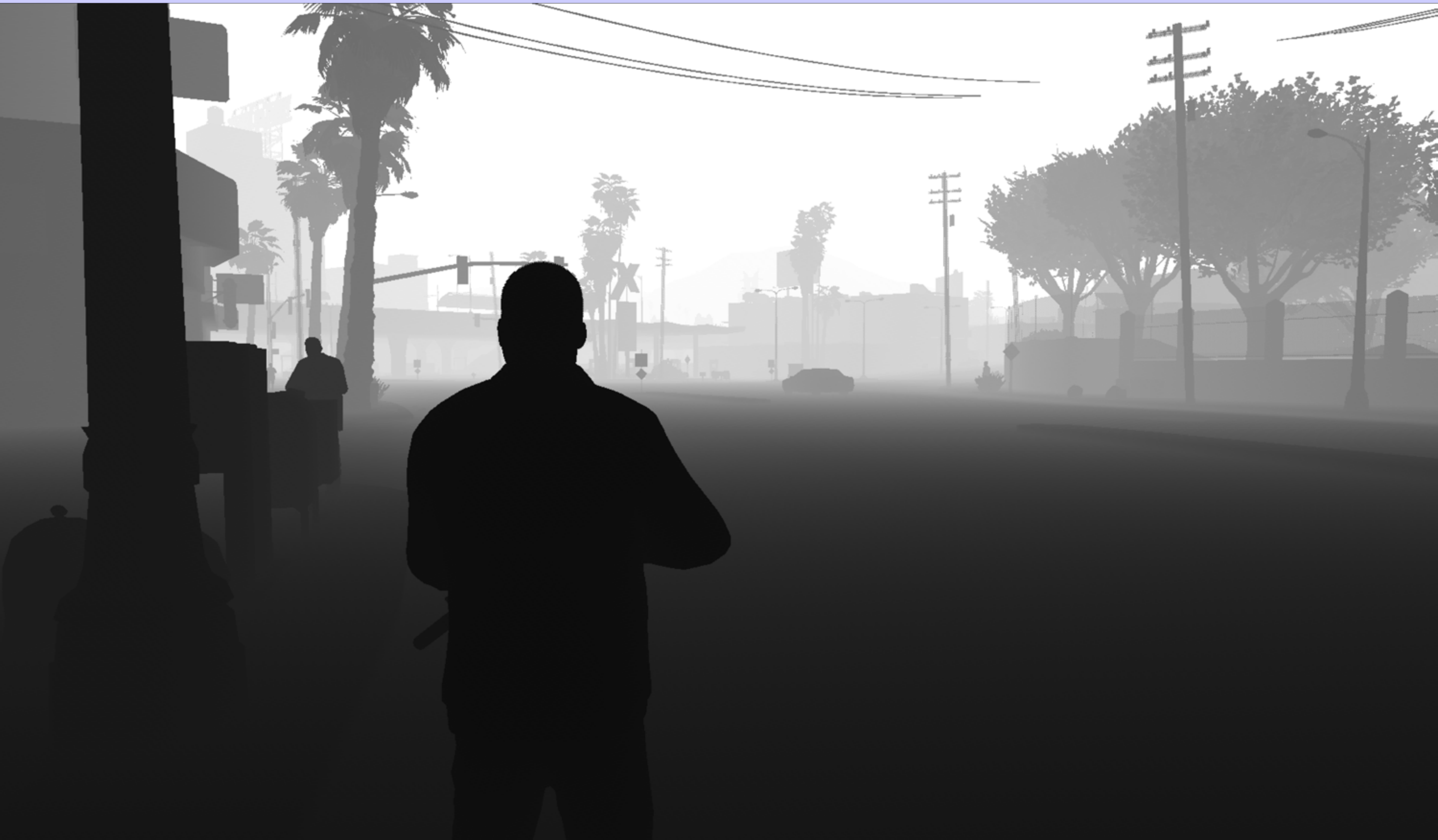
The color buffer



The z-buffer
(or depth buffer)

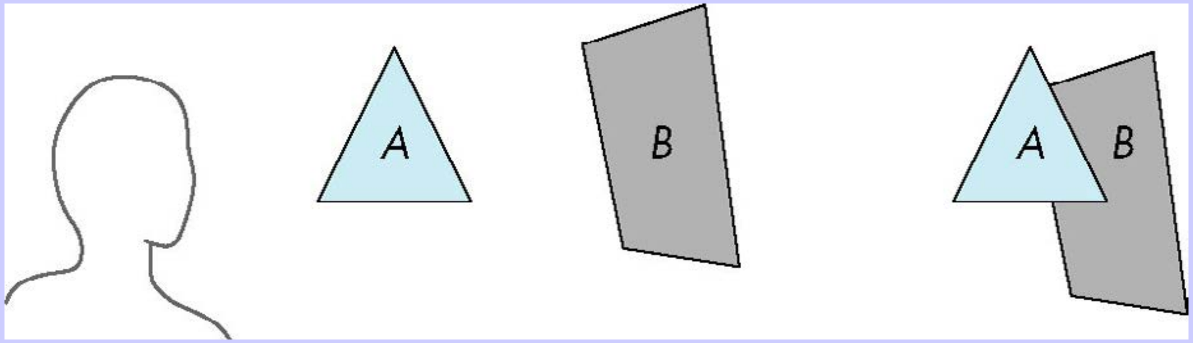
The RASTERIZER

Z-buffer



Painter's Algorithm

- Render polygons a back to front order so that polygons behind others are simply painted over



B behind A as seen by viewer

Fill B then A

- Requires ordering of polygons first
  - O(n log n) calculation for ordering
  - Not every polygon is either in front or behind all other polygons

I.e., : Sort all triangles and render them back-to-front.

Z-Buffer Algorithm

- Use a buffer called the z or depth buffer to store the depth of the closest object at each pixel found so far
- As we render each polygon, compare the depth of each new fragment, d_{new}, to depth in z buffer, d_{zb}
- If $d_{new} < d_{zb}$ (new fragment is closer to cam), replace pixel's color and z-buffer value.

The RASTERIZER double-buffering

Application
Geometry
Rasterizer

- The monitor displays one image at a time
- Top of screen – new image

  Bottom – old image

  No control of split position
- And even worse, we often clear the screen before generating a new image
- A better solution is ”double buffering”
  - (Could instead keep track of rasterpos and vblank).

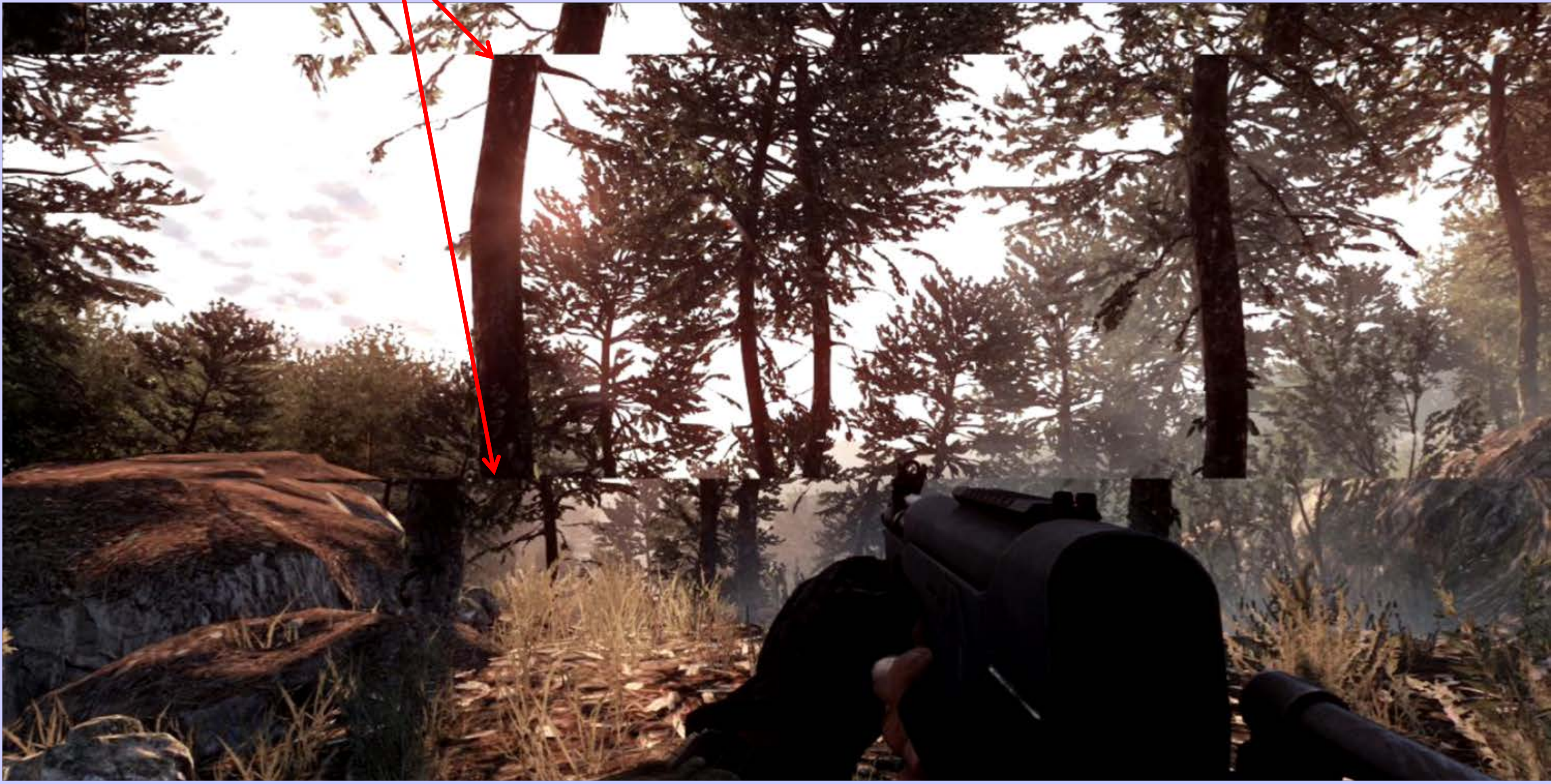
The RASTERIZER double-buffering

- Use two buffers: one front and one back
- The front buffer is displayed
- The back buffer is rendered to
- When new image has been created in back buffer, swap front and back

Remedies screen-tearing problems but not completely…

Screen Tearing

Swapping
back/front buffers



vblank

Screen tearing is solved by using V-Sync.
V-Sync: swap front/back buffers during vertical blank (vblank) instead.

Screen Tearing

- Despite the gorgeous graphics seen in many of today's games, there are still some highly distracting artifacts that appear in gameplay despite our best efforts to suppress them. The most jarring of these is screen tearing. Tearing is easily observed when the mouse is panned from side to side. The result is that the screen appears to be torn between multiple frames with an intense flickering effect. Tearing tends to be aggravated when the framerate is high since a large number of frames are in flight at a given time, causing multiple bands of tearing.
- **Vertical sync (V-Sync) is the traditional remedy to this problem**, but as many gamers know, V-Sync isn't without its problems. The main problem with V-Sync is that when the framerate drops below the monitor's refresh rate (typically 60 fps), the framerate drops disproportionately. For example, dropping slightly below 60 fps results in the framerate dropping to 30 fps. This happens because the monitor refreshes at fixed internals (although an LCD doesn't have this limitation, the GPU must treat it as a CRT to maintain backward compatibility) and V-Sync forces the GPU to wait for the next refresh before updating the screen with a new image. This results in notable stuttering when the framerate dips below 60, even if just momentarily.

What is important:

- Understand the Application-, Geometry- and Rasterization Stage
- Correlation to hardware
- Z-buffering, double buffering, screen tearing

BONUS

Simple Application...

OLD WAY

OpenGL 1.1

```
#ifdef WIN32
#include <windows.h>
#endif

#include <GL/glut.h>                    // This also includes gl.h

static void drawScene(void)
{
    glColor3f(1,1,1);


    glBegin(GL_POLYGON);
        glVertex3f( 4.0, 0, 4.0);
        glVertex3f( 4.0, 0,-4.0);
        glVertex3f(-4.0, 0,-4.0);
    glEnd();
}
```

Usually this and next 2 slides are put in the same file main.cpp

BONUS
Old way
OpenGL 1.1

Simple Application

```
void display(void)
{
      glClearColor(0.2, 0.2, 0.8, 1.0);             // Set clear color
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clears the color buffer
                                                                and the z-buffer
      int w = glutGet((GLenum)GLUT_WINDOW_WIDTH);
      int h = glutGet((GLenum)GLUT_WINDOW_HEIGHT);
      glViewport(0, 0, w, h);                       // Set viewport

      glMatrixMode(GL_PROJECTION);                  // Set projection matrix
      glLoadIdentity();
      gluPerspective(45.0,w/h, 0.2, 10000.0);       // FOV, aspect ratio, near, far

      glMatrixMode(GL_MODELVIEW);                   // Set modelview matrix
      glLoadIdentity();

      gluLookAt(10, 10, 10,                         // look from
                0, 0, 0,                            // look at
                0, 0, 1);                           // up vector

      drawScene();
      glutSwapBuffers();  // swap front and back buffer. This frame will now been displayed.
}
```

Changing Color per Vertex

BONUS
Old way
OpenGL 1.1

```
static void drawScene(void)
{
   // glColor3f(1,1,1);
   glBegin(GL_POLYGON);
         glColor3f(1,0,0);
         glVertex3f( 4.0, 0, 4.0);

         glColor3f(0,1,0);
         glVertex3f( 4.0, 0,-4.0);

         glColor3f(0,0,1);
         glVertex3f(-4.0, 0,-4.0);
   glEnd();
}
```