TDA361 – Computer Graphics

Teacher: Ulf Assarsson

Assistants:
Erik Sintorn (postdoc)
Viktor Kämpe (PhD student)
Dan Dolonius (Autodesk)
Vasileios Golematis (student)
Johan Bowald (student)

This Course

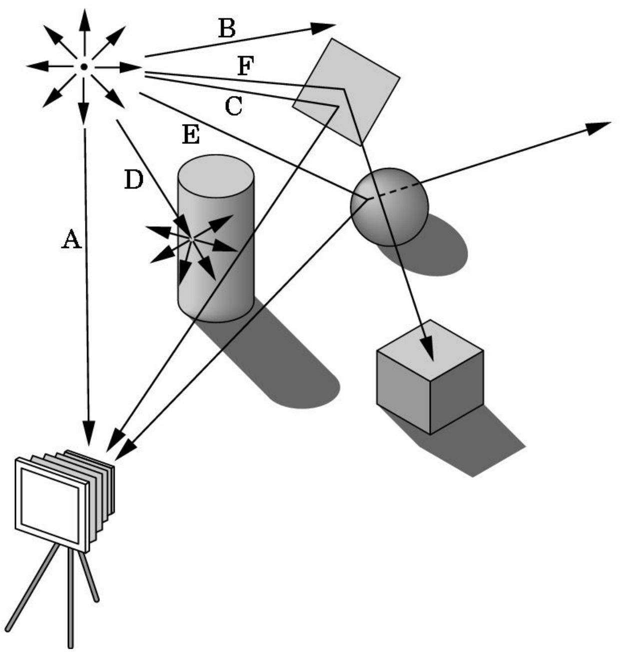
- Algorithms!

Understanding of Ray Tracing

Real-time Rendering

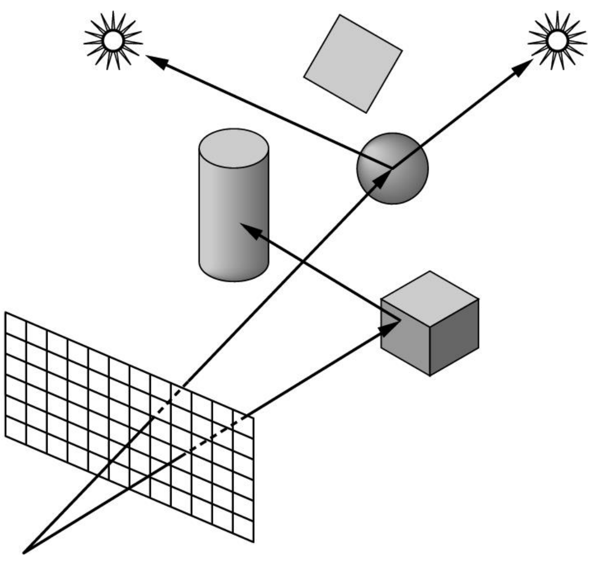
Tracing Photons

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

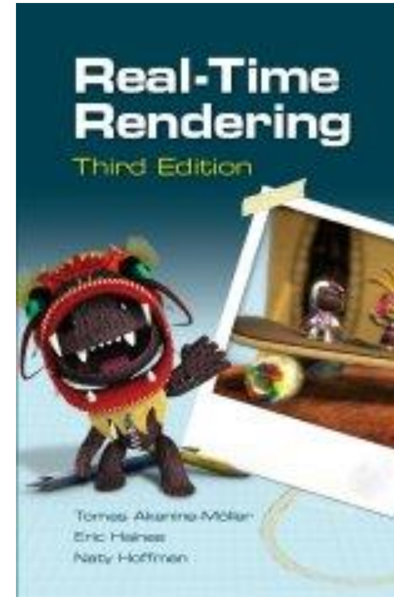
- **Ray tracing**: follow rays of light from center of projection until they either are absorbed by objects or go off to infinity
  - Can handle global effects
    - Multiple reflections
    - Translucent objects
  - Faster but still slow



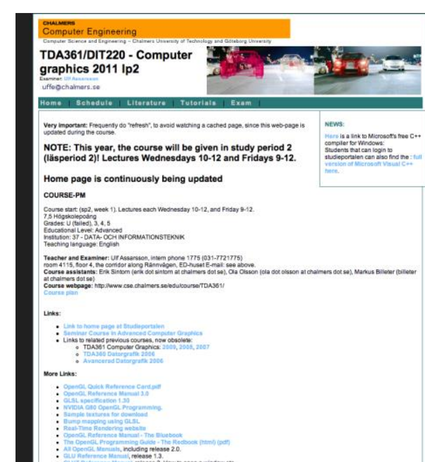
I’m here to help…

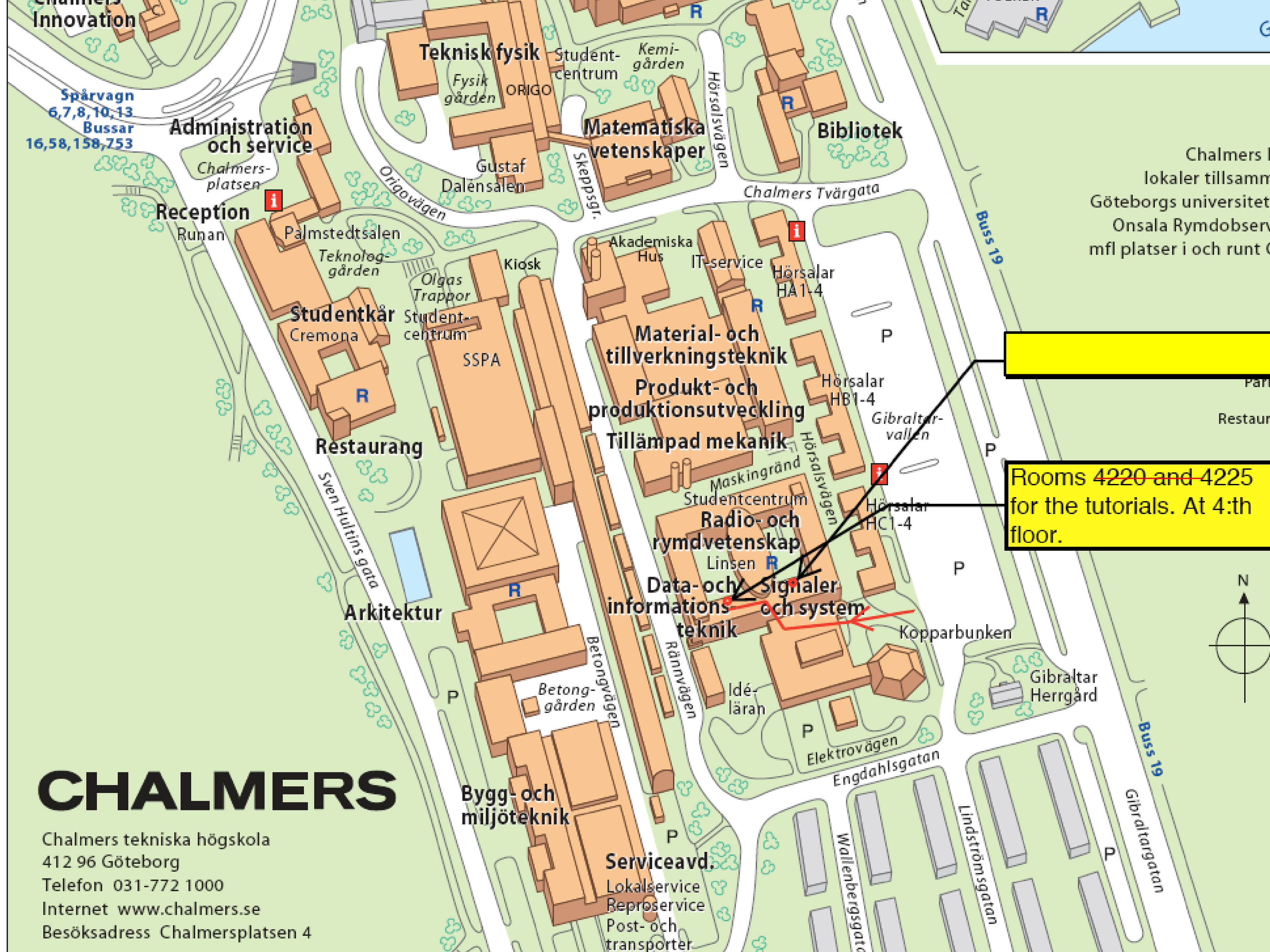
1. I am located in room 4115 in ”EDIT-huset”
2. Email: uffe at chalmers dot se
3. Phone: 031-772 1775 (office)
4. Course assistant:
   1. kampe at chalmers dot se (Viktor Kämpe)
   2. gusdolod at student dot gu dot se (Dan Dolonius)

Course Info



- Study Period 2 (lp2)
- Real Time Rendering, 3rd edition
  - Available on Cremona
- Schedule:
  - Tues 10-12 + Fri 9-12 + Mon 10/11 13-16 + Mon 24/11 13-16
    - 14 lectures in total, ~2 / week
  - Labs: 17-21 everyday, 13-17 Tuesdays and Wednesdays, 9-12 Thursdays
- Homepage:
  - Google “TDA361” or
  - “Computer Graphics Chalmers”

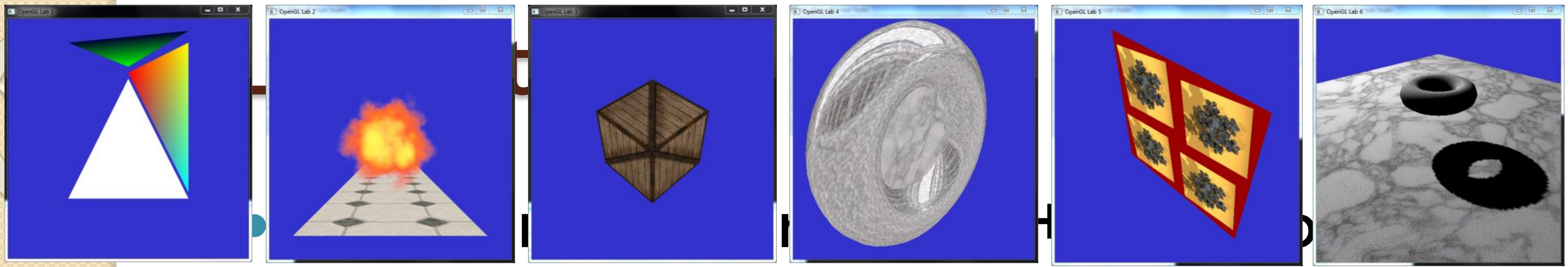



Innovation
Teknisk fysik
Fysik gården
ORIGO
Student- centrum
Kemi- gården
Hörsalsvägen
Bibliotek
Spårvagn 6,7,8,10,13
Bussar 16,58,158,753
Administration och service
Chalmers- platsen
Matematiska vetenskaper
Gustaf Dalénsalen
Origovägen
Skeppsgr.
Chalmers Tvärgata
Reception
Runan
Palmstedtsalen
Teknolog- gården
Olgas Trappor
Kiosk
Akademiska Hus
IT-service
Hörsalar HA1-4
Buss 19
Chalmers
lokaler tillsamm
Göteborgs universitet
Onsala Rymdobserv
mfl platser i och runt
Studentkår
Cremona
Student- centrum
SSPA
Material- och tillverkningsteknik
Produkt- och produktionsutveckling
Hörsalar HB1-4
Gibraltar- vallen
Tillämpad mekanik
Restaurang
Maskingränd
Studentcentrum
Hörsalar HC1-4
Radio- och rymdvetenskap
Linsen
Rooms ~~4220 and 4225~~ for the tutorials. At 4:th floor.
Sven Hultins gata
Arkitektur
Data- och informationsteknik
Signaler och system
Kopparbunken
N
Betongvägen
Rännvägen
Betong- gården
Idé- läran
Gibraltar Herrgård
Elektrovägen
Engdahlsgatan
CHALMERS
Bygg- och miljöteknik
Serviceavd.
Lokalservice
Reproservice
Post- och transporter
Wallenbergsgatan
Lindströmsgatan
Gibraltargatan
Chalmers tekniska högskola
412 96 Göteborg
Telefon 031-772 1000
Internet www.chalmers.se
Besöksadress Chalmersplatsen 4

Laborations

- 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
  - Render engine (e.g for a game)

    or
  - Path tracer




3D World Tutorial - SOLUTION

Tutorials

- Rooms 4225
  - Or your favorite place/home
- 4th floor EDIT-building
- EntranceCards (inpasseringskort)
  - Automatically activated for all of you that are course registered and have a CTH/GU-entrance card (inpasseringskort)
- Recommended to do the tutorials in groups (Labgrupper) of 2 and 2

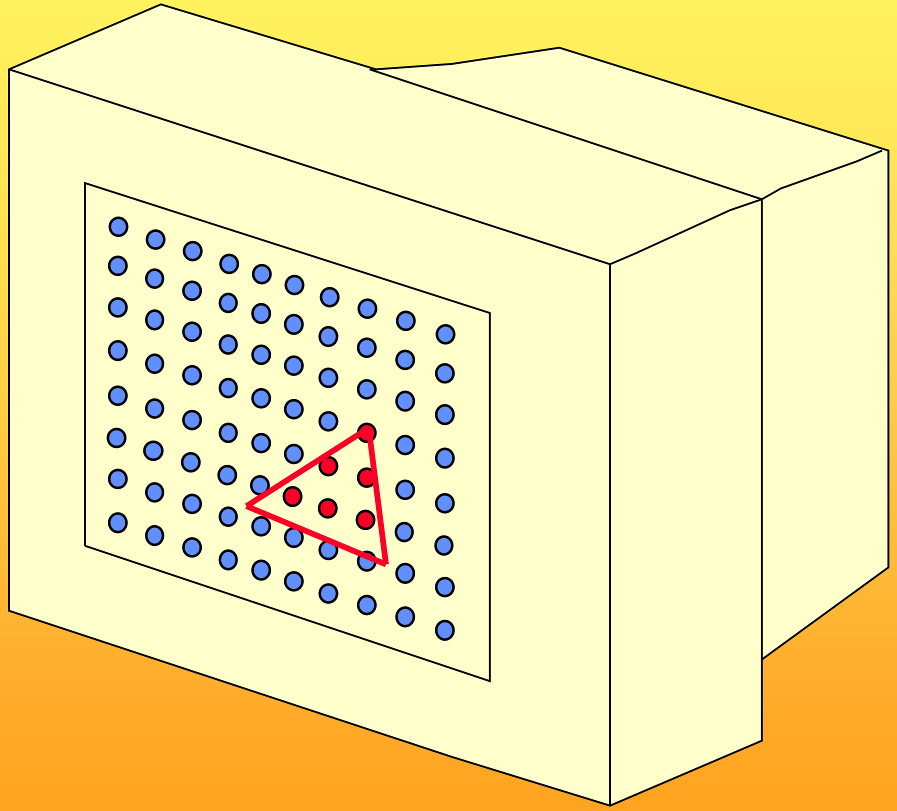
Overview of the Graphics Rendering Pipeline and OpenGL

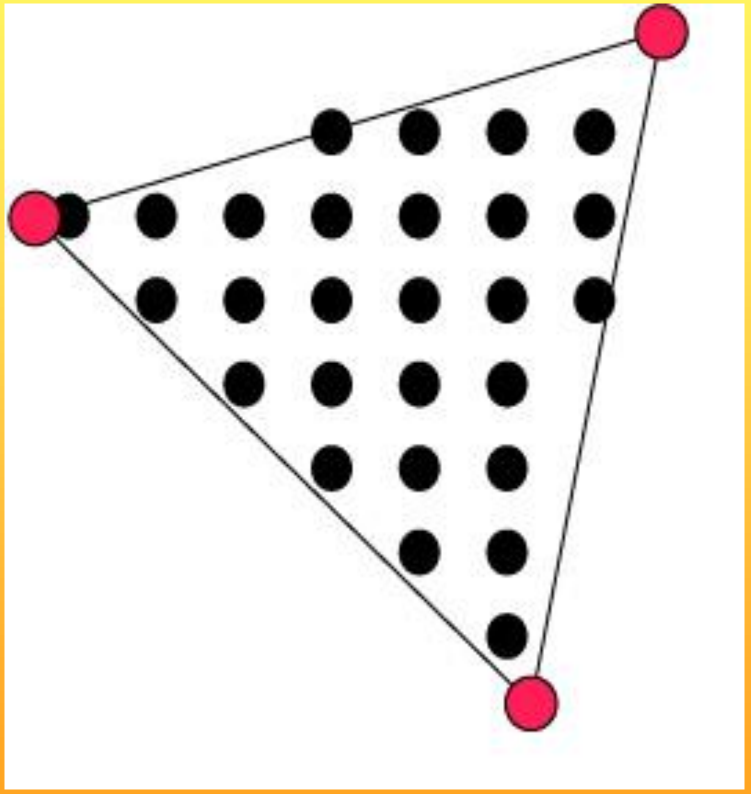
CHALMERS

3D Graphics

Ulf Assarsson

The screen consists of many pixels

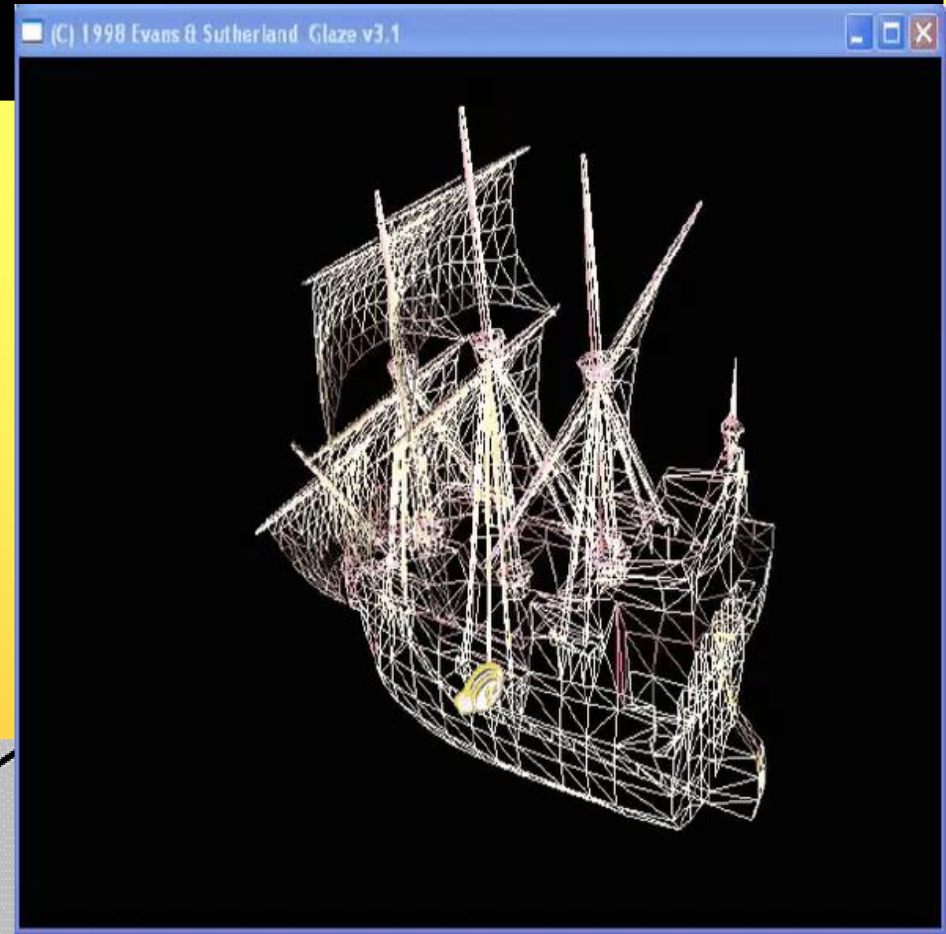


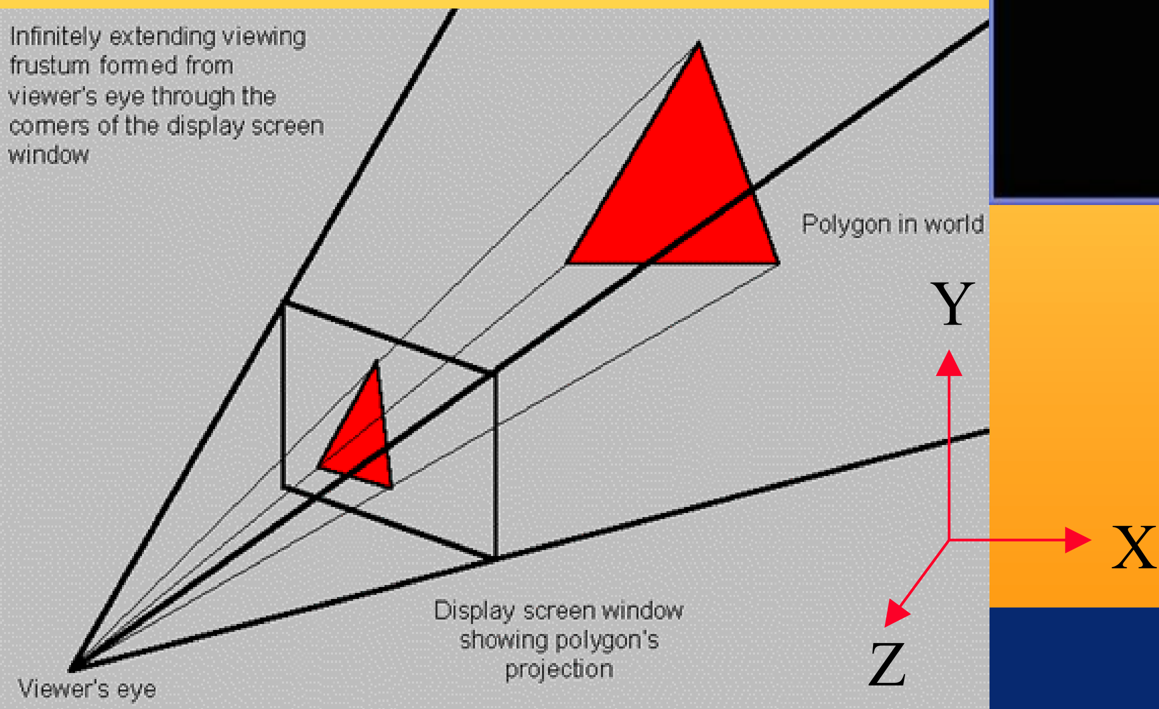



Twin Frozr
msi
HDMI

3D-Rendering

- Objects are often made of triangles
- x,y,z- coordinate for each vertex



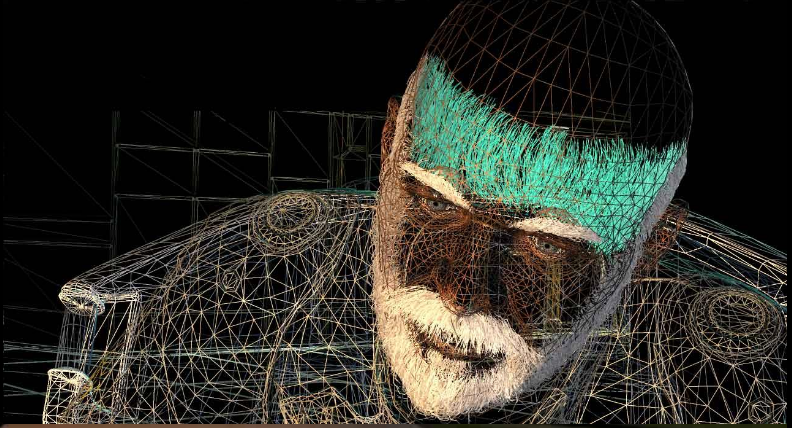


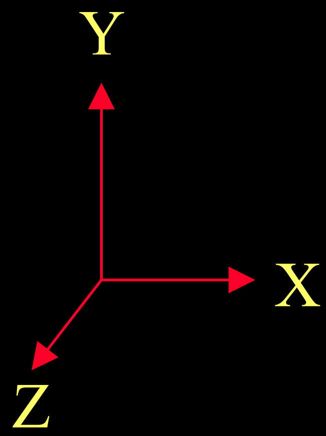
Why only triangles?

4D Matrix Multiplication

$$\begin{bmatrix} s_x & \bullet & \bullet & t_x \\ \bullet & s_y & \bullet & t_y \\ \bullet & \bullet & s_z & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

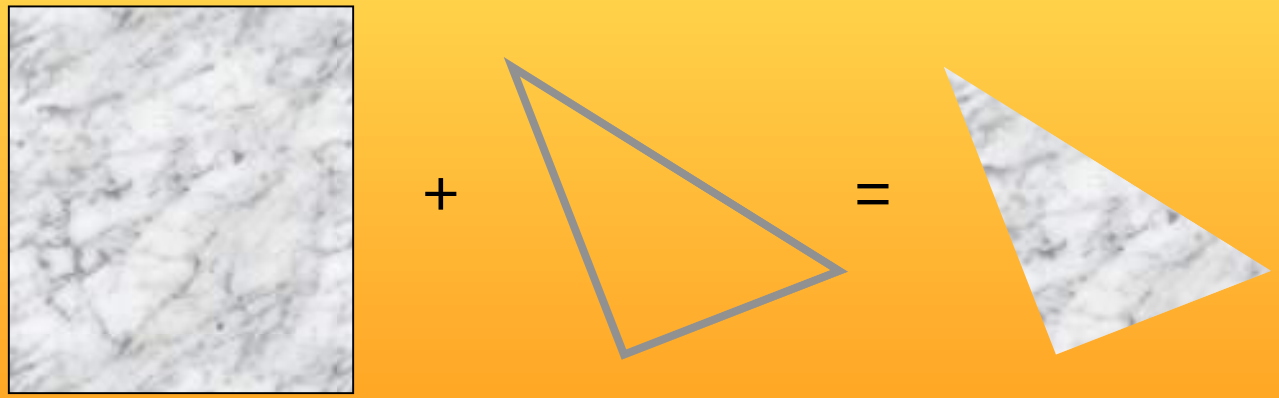
Real-Time Rendering




Y
X
Z

Textures

- One application of texturing is to ”glue” images onto geometrical object

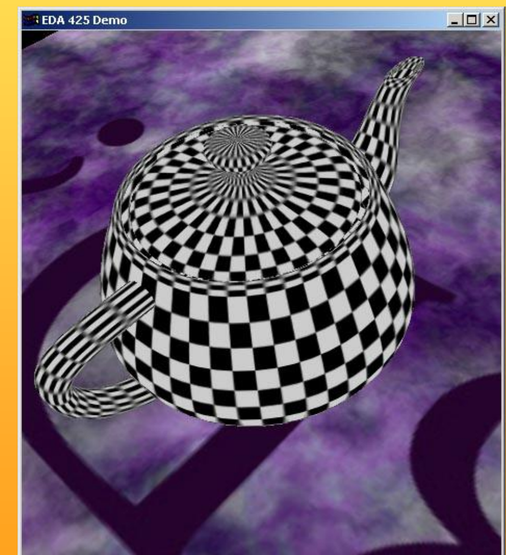


Texturing: Glue images onto geometrical objects

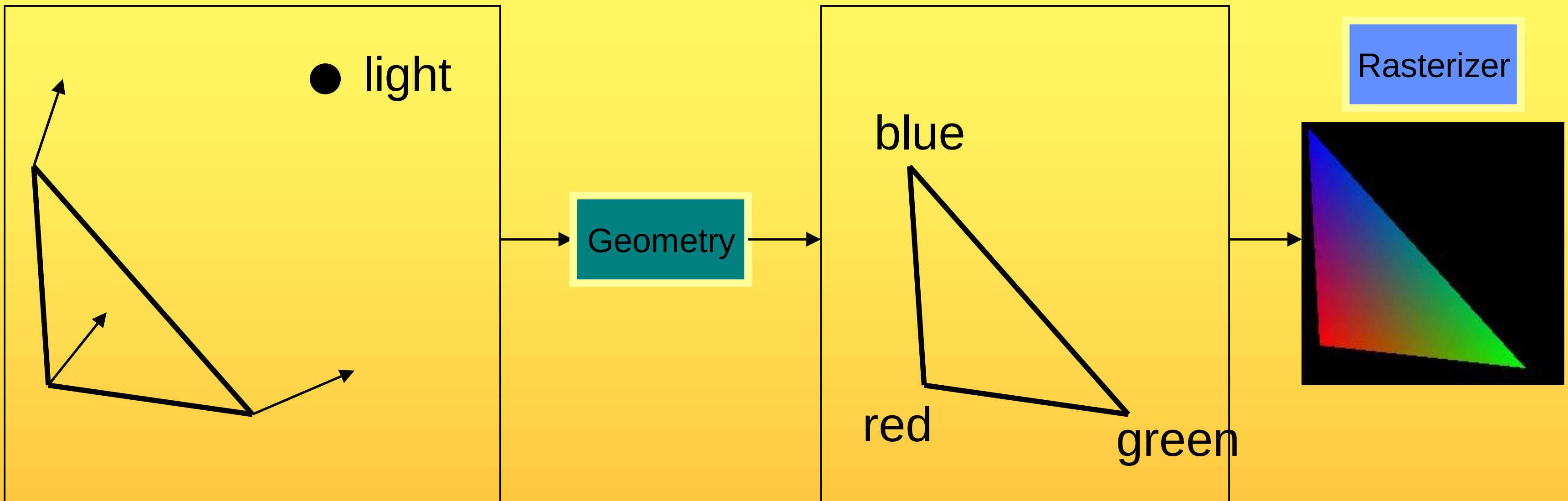
- Purpose: more realism, and this is a cheap way to do it

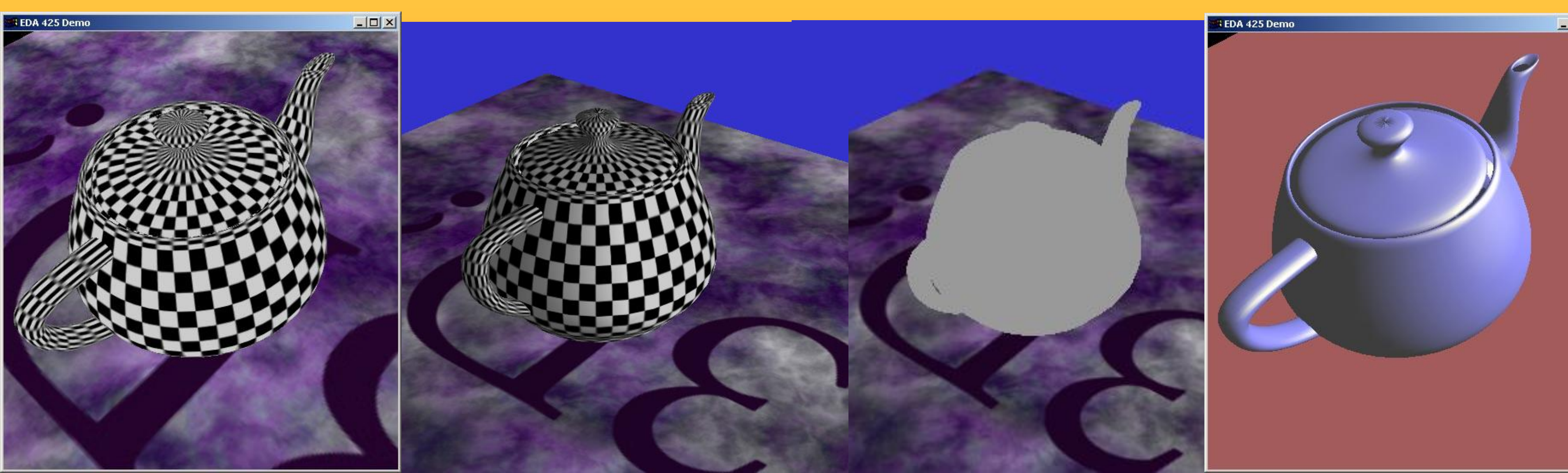
+

=



Lighting computation per triangle vertex

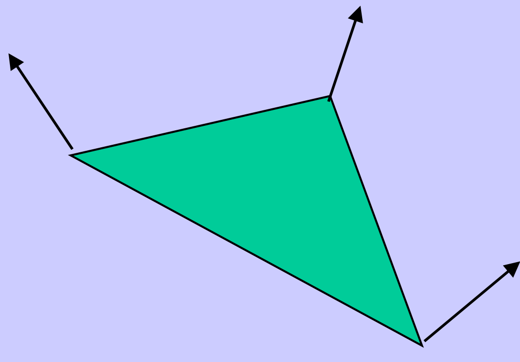




The Graphics Rendering Pipeline

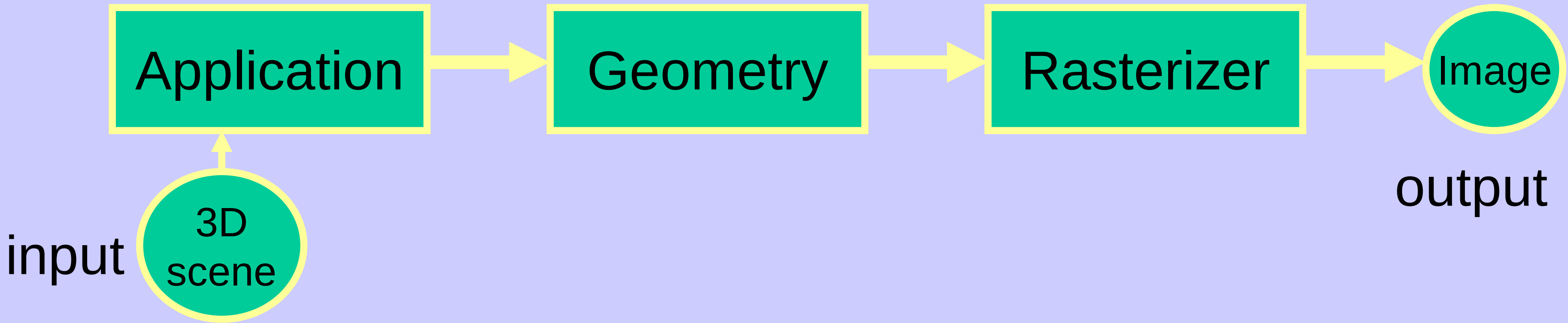
You say that you render a *”3D scene”*, but what is it?

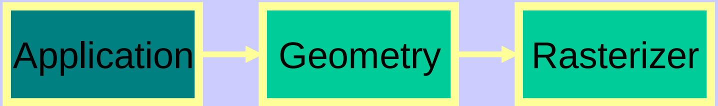
- First, of all to take a picture, it takes a camera – a virtual one.
  - Decides what should end up in the final image
- A 3D scene is:
  - Geometry (triangles, lines, points, and more)
  - Light sources
  - Material properties of geometry
    - Colors, shader code ,
    - Textures (images to glue onto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



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

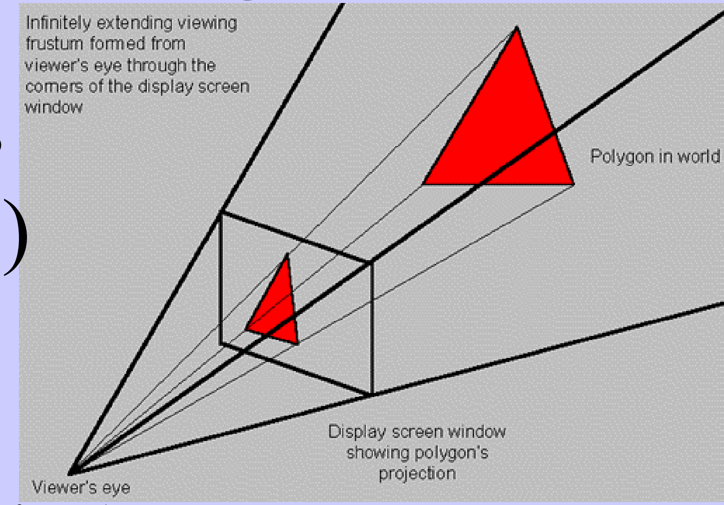




The APPLICATION stage

- 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

The GEOMETRY stage

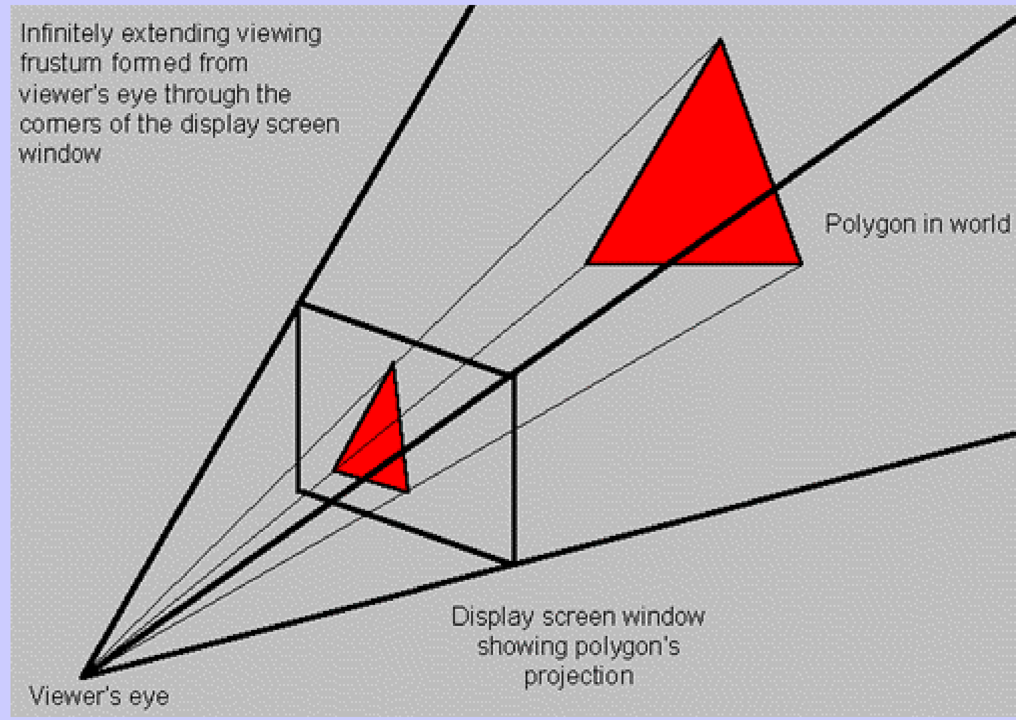


- Task: ”geometrical” operations on the input data (e.g. triangles)
- Allows:
  - Move objects (matrix multiplication)
  - Move the camera (matrix multiplication)
  - Lighting computations per triangle vertex
  - Project onto screen (3D to 2D)
  - Clipping (avoid triangles outside screen)
  - Map to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

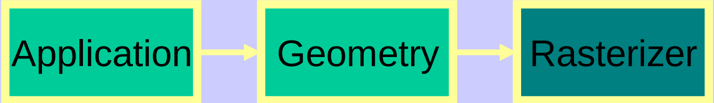
Application → Geometry → Rasterizer

The GEOMETRY stage

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

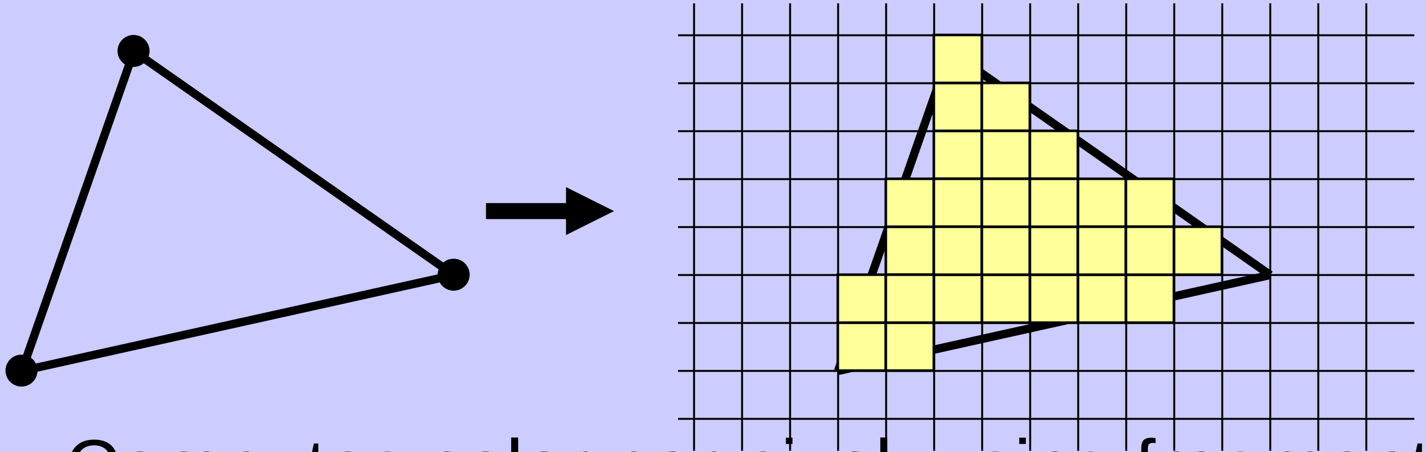


- (Instances)
- Vertex Shader
  - A program executed per vertex
    - Transformations
    - Projection
    - E.g., color per vertex
- Clipping
- Screen Mapping



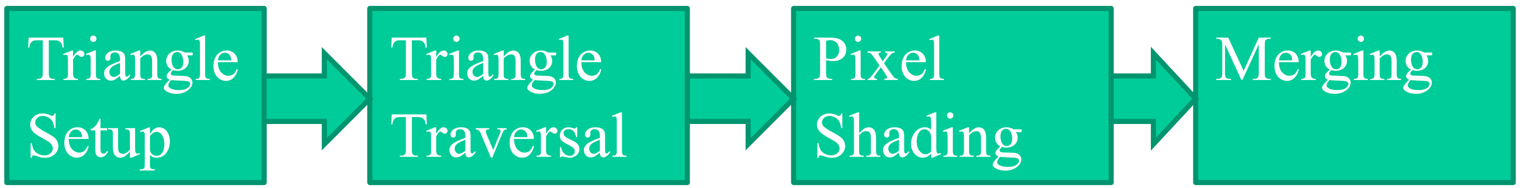
The RASTERIZER stage

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



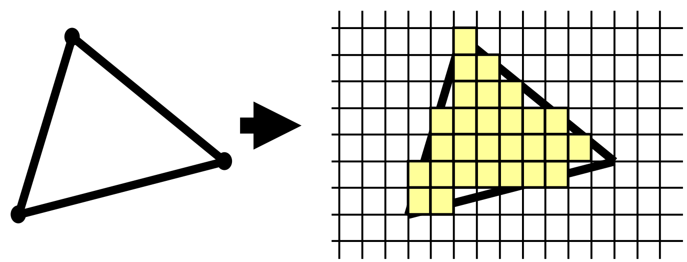
- Computes color per pixel, using fragment shader (=pixel shader)
  - textures, (light sources, normal), colors and various other per-pixel operations
- And visibility is resolved here: sorts the primitives in the z-direction

The rasterizer stage



Triangle Setup:

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



Triangle Traversal

- Scan conversion

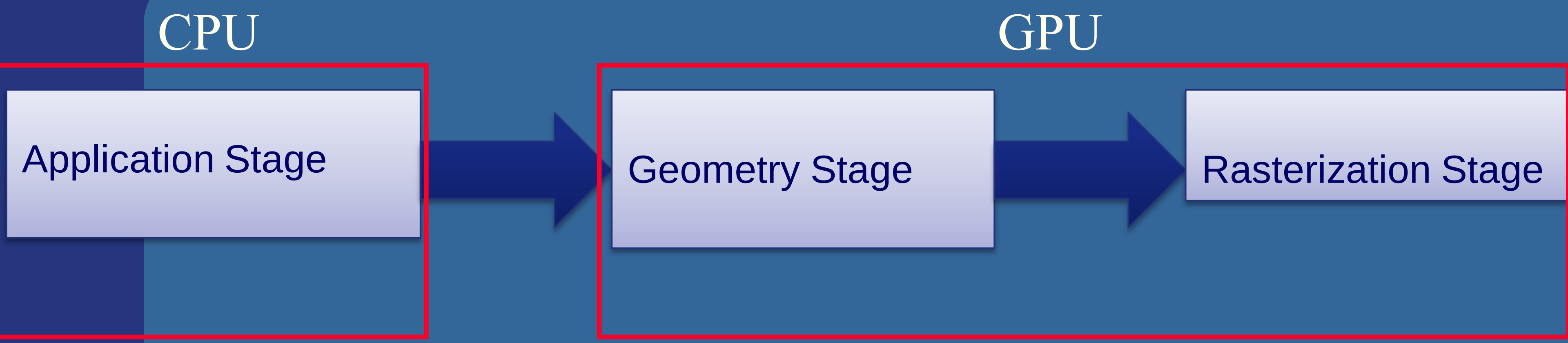
Pixel Shading

- Compute pixel color

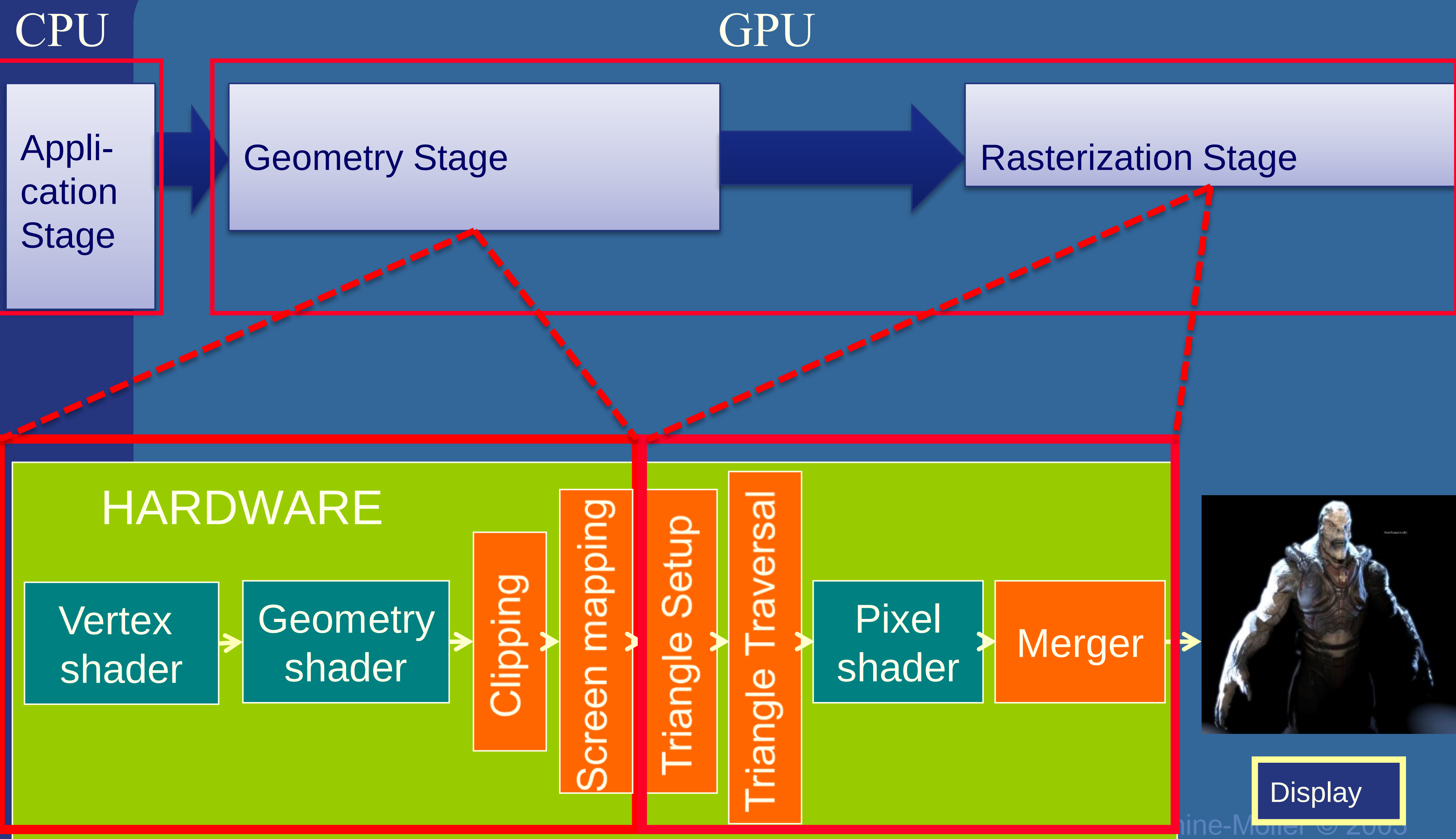
Merging:

- output color to screen

Rendering Pipeline and Hardware



Rendering Pipeline and Hardware

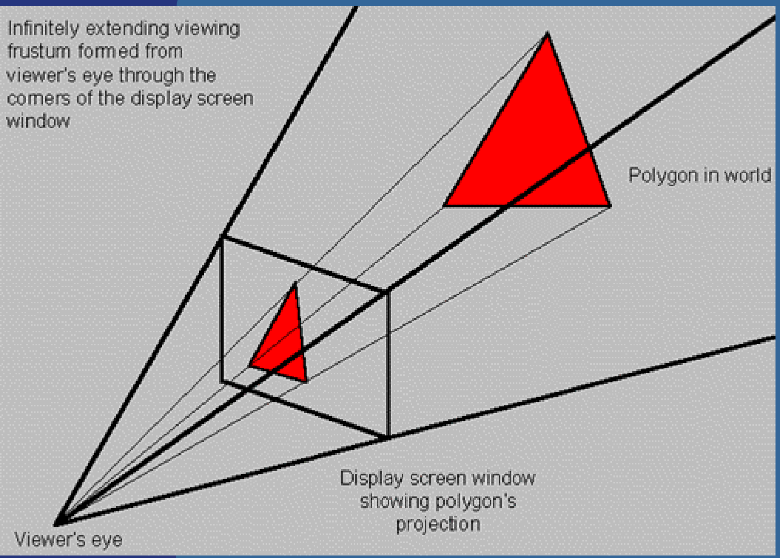


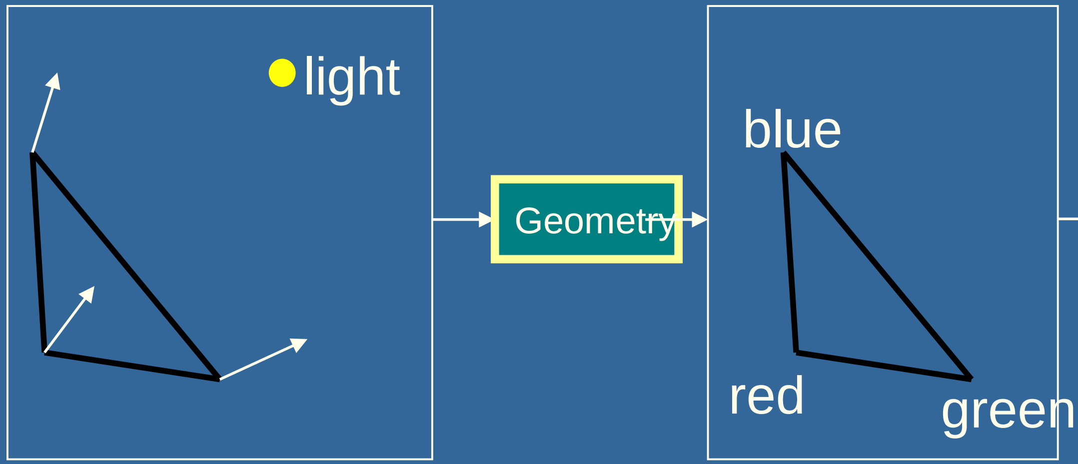
Hardware design

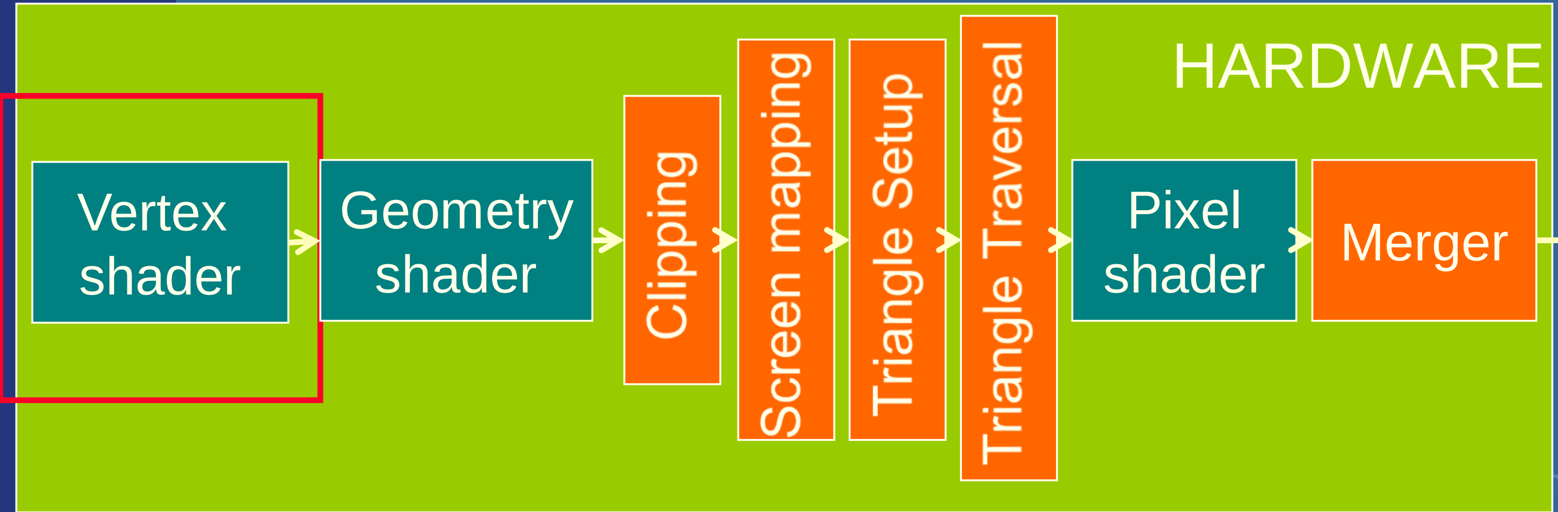
Geometry Stage

Vertex shader:

- Lighting (colors)
- Screen space positions







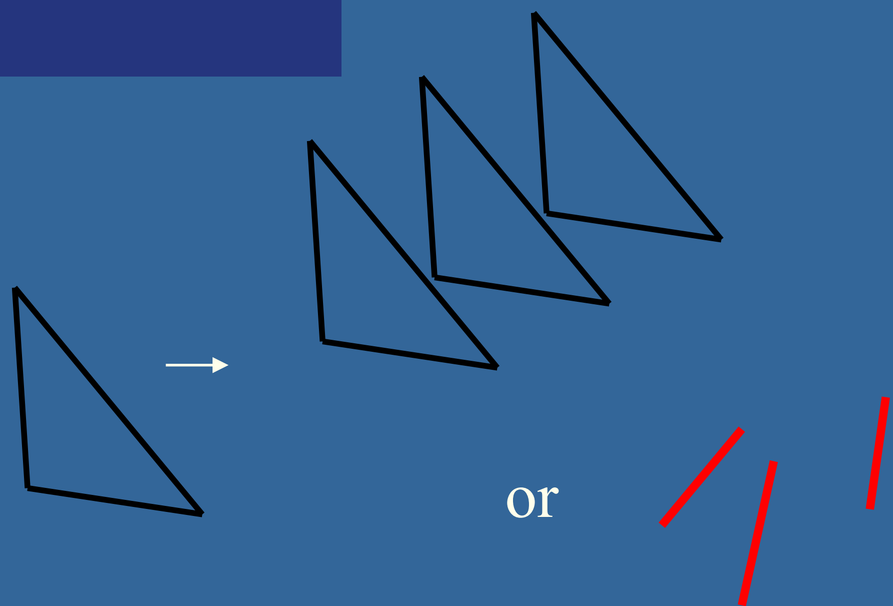
Display

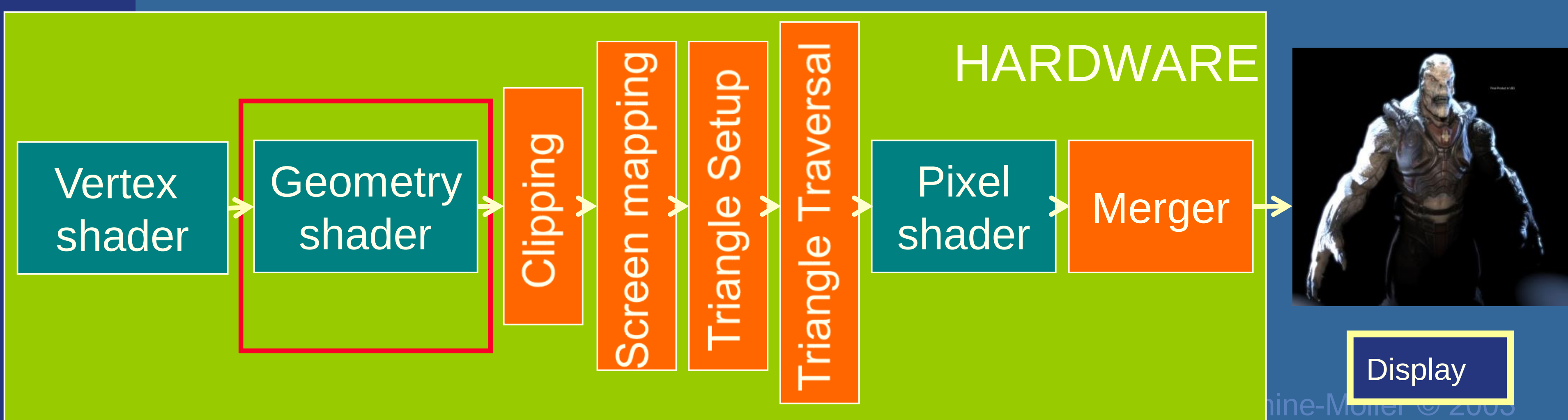
Hardware design

Geometry Stage

Geometry shader:

- One input primitive
- Many output primitives

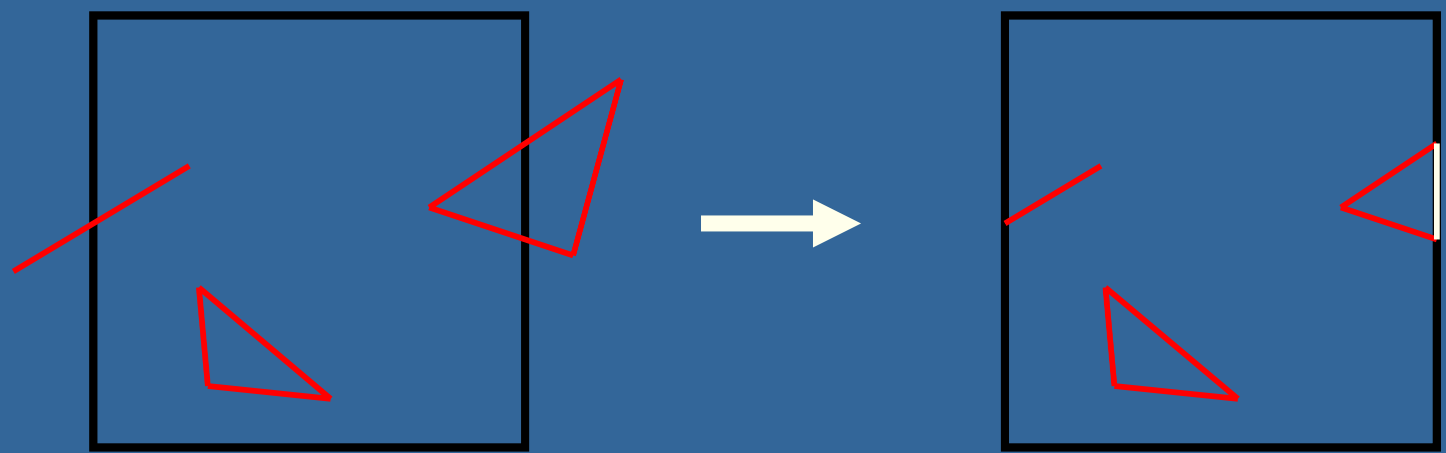


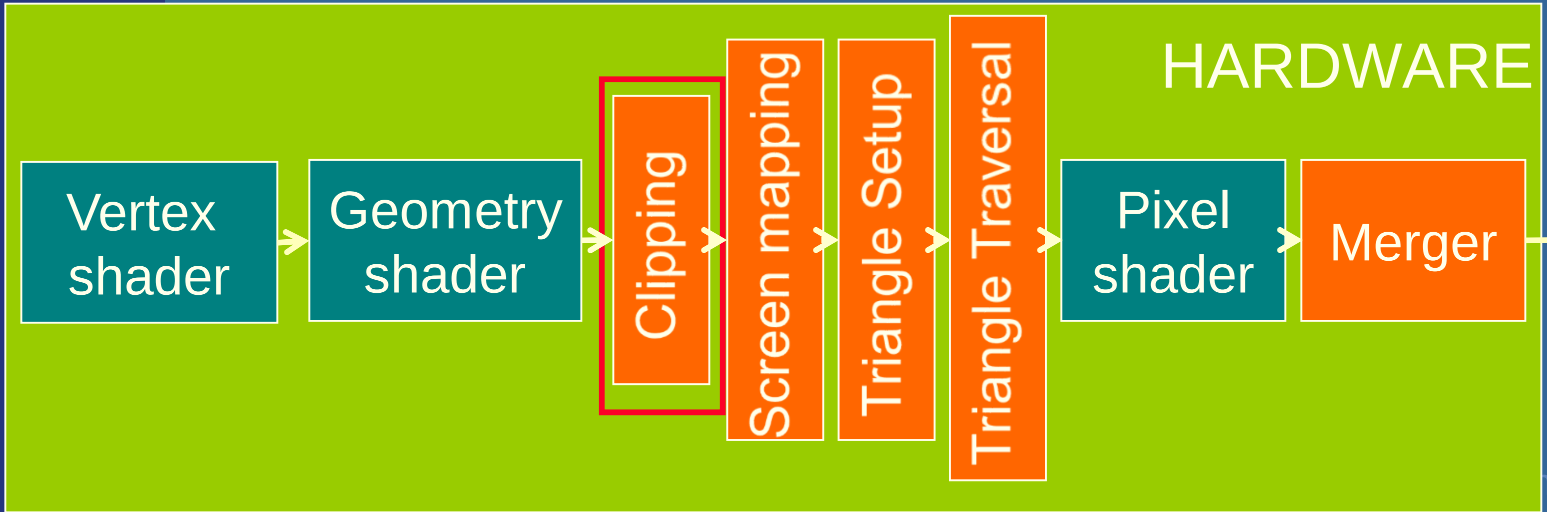


Hardware design

Geometry Stage

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





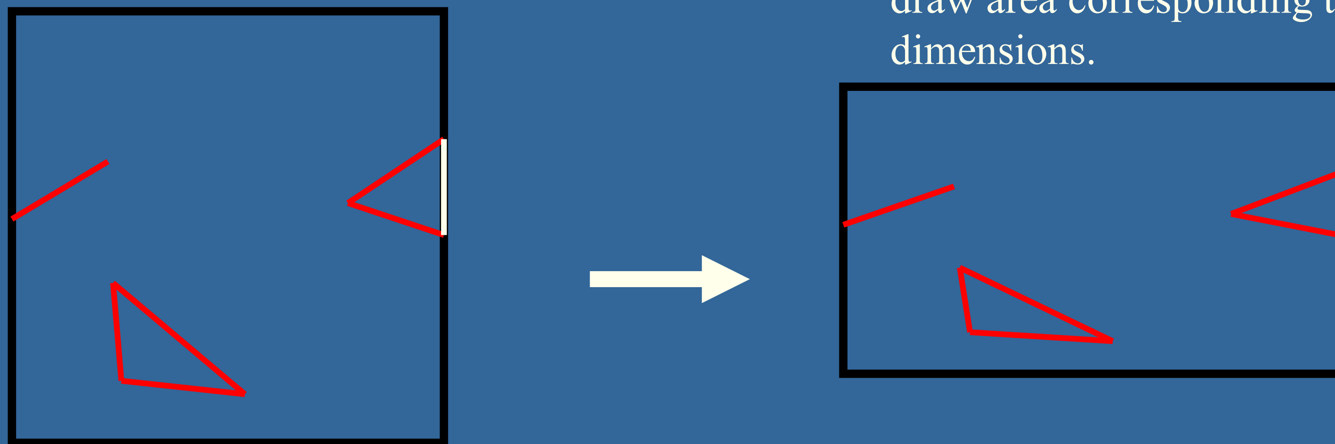
Display

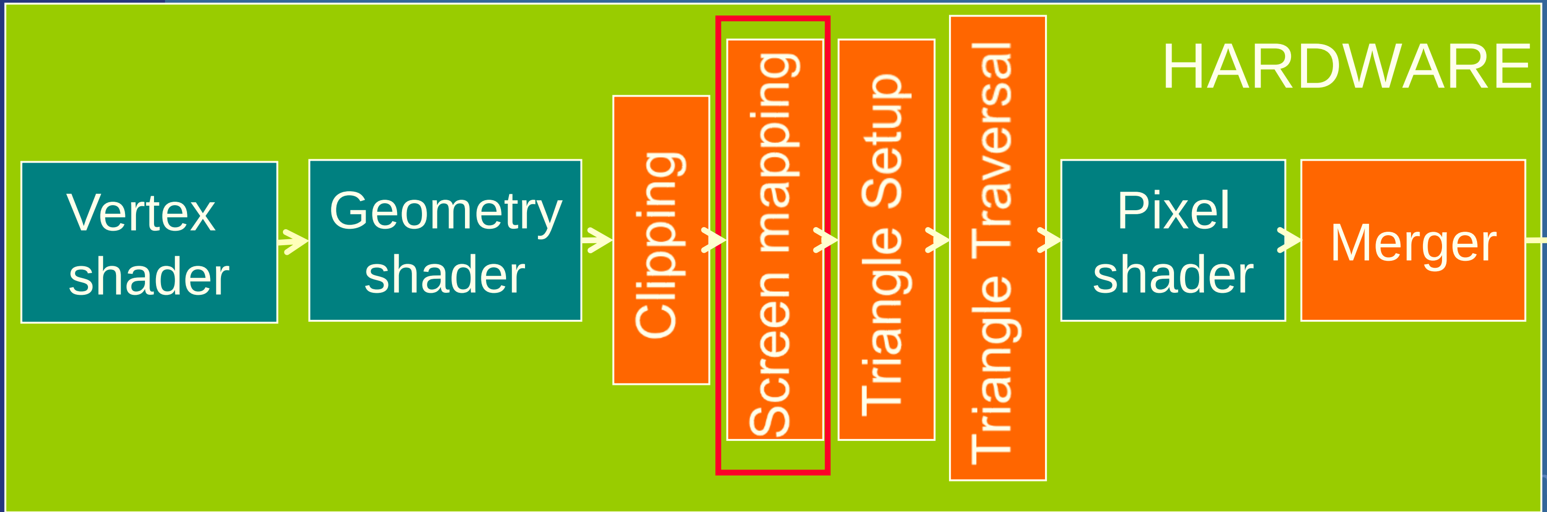
Hardware design

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.



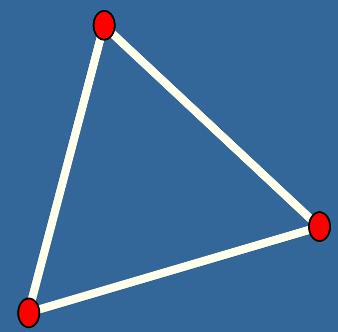


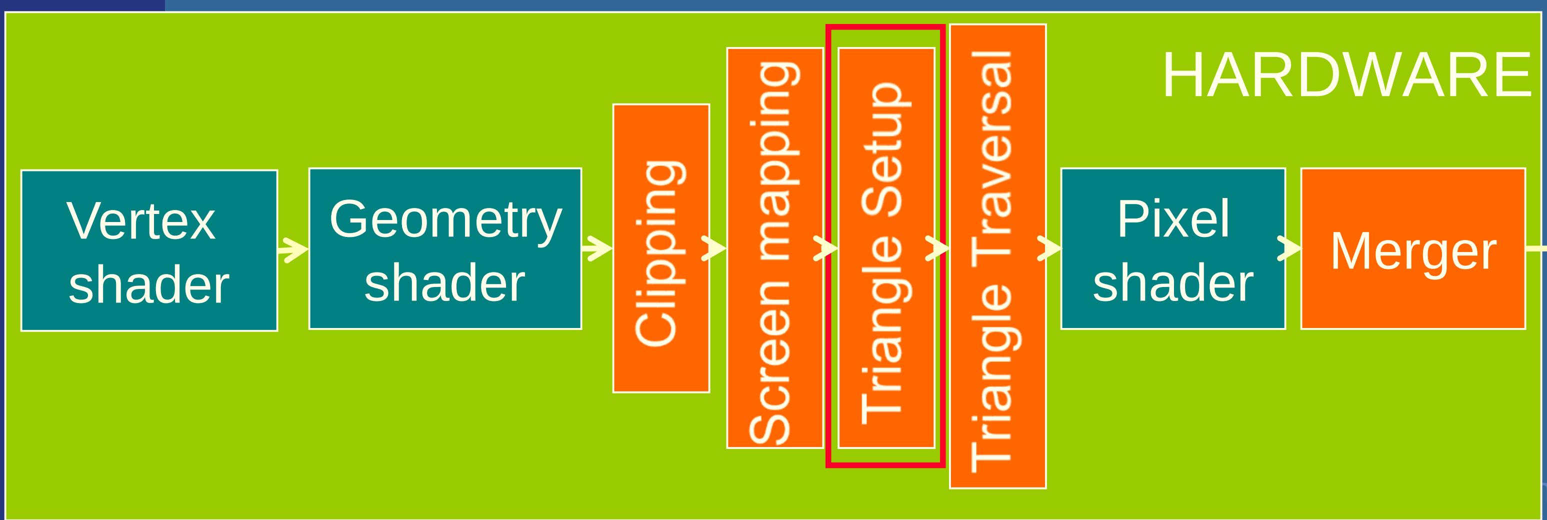
Display

Hardware design

Rasterizer Stage

Collects three vertices into one triangle



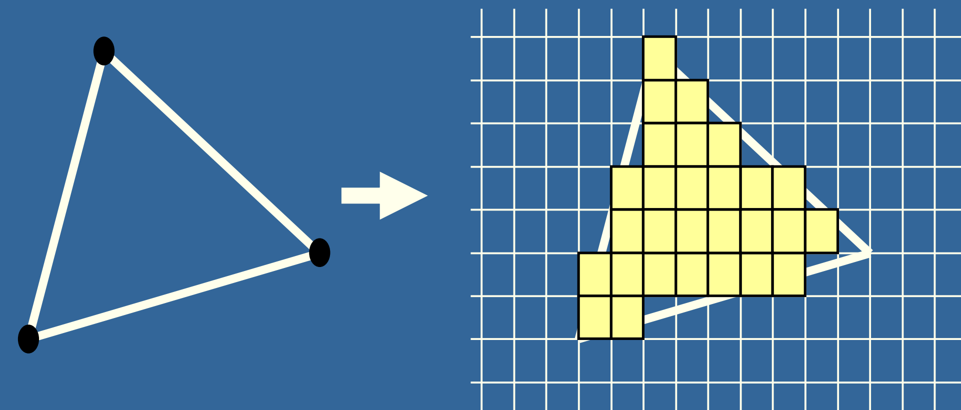


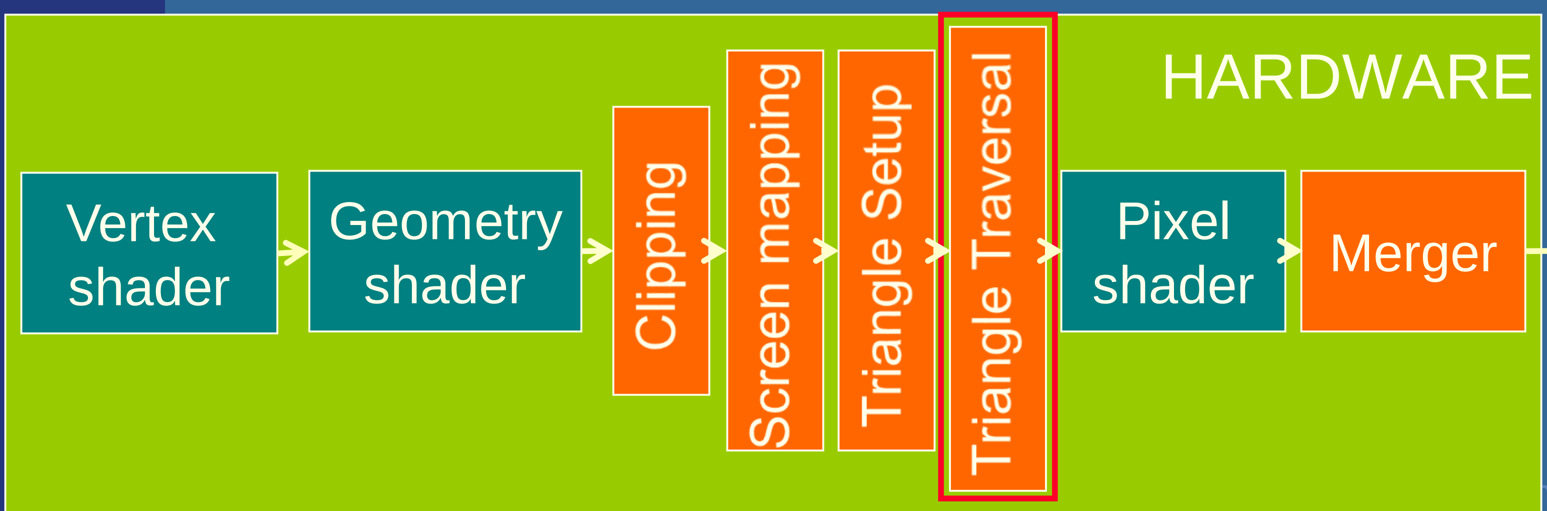
Display

Hardware design

Rasterizer Stage

Creates the fragments/pixels for the triangle

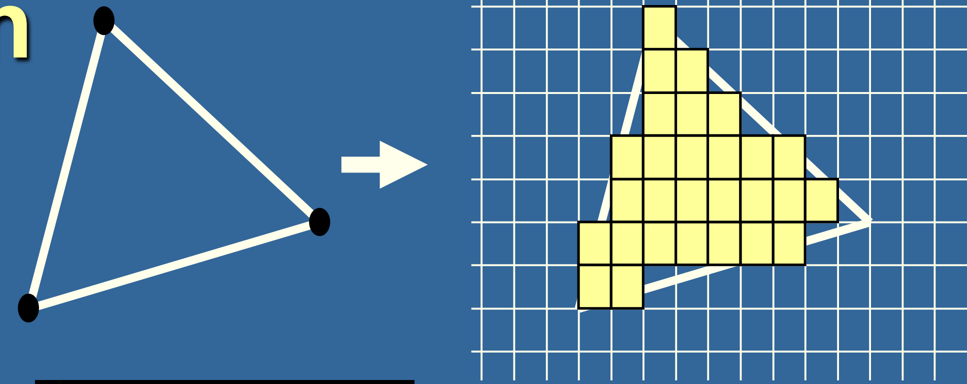


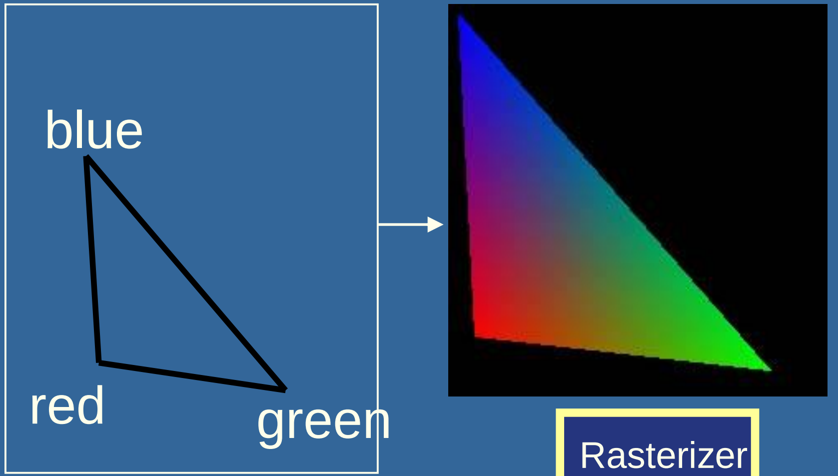


Display

Hardware design

Rasterizer Stage

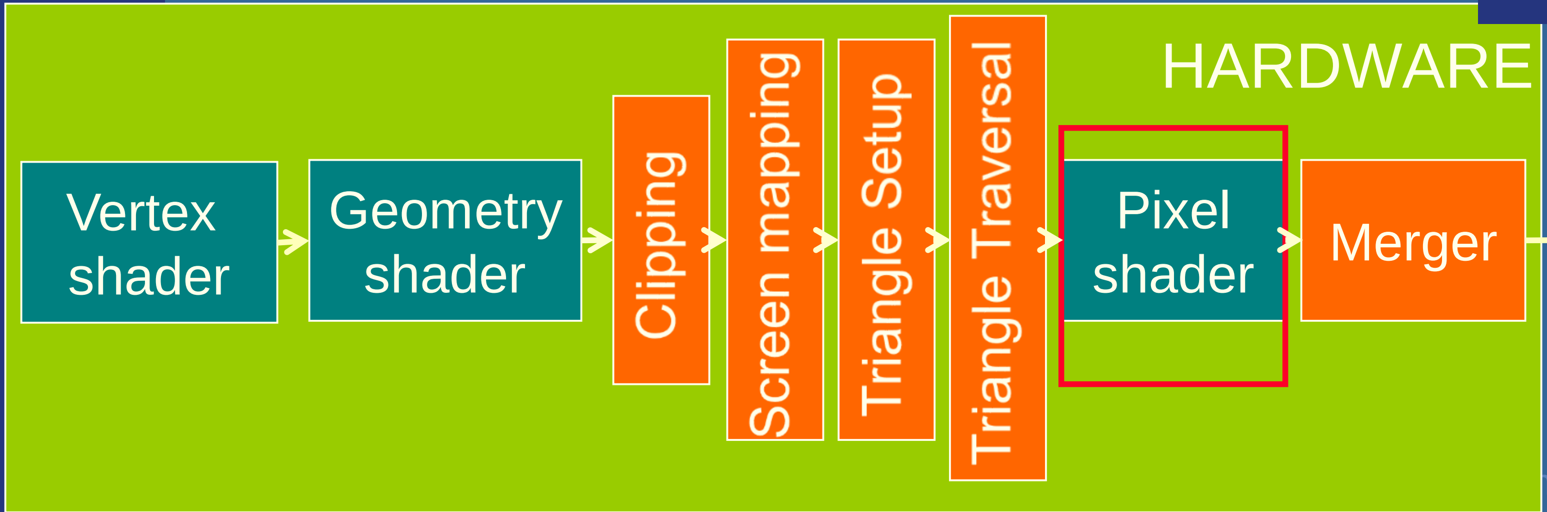




Rasterizer

Pixel Shader:
Compute color using:

- Textures
- Interpolated data (e.g. Colors + normals) from vertex shader



Display

Hardware design

Rasterizer Stage

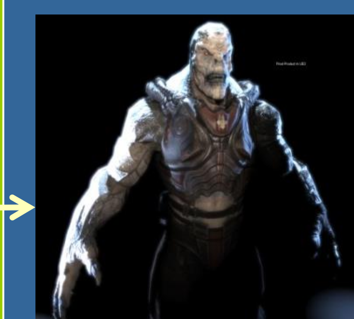
The merge units update the frame buffer with the pixel's color

Frame buffer:

- Color buffers
- Depth buffer
- Stencil buffer

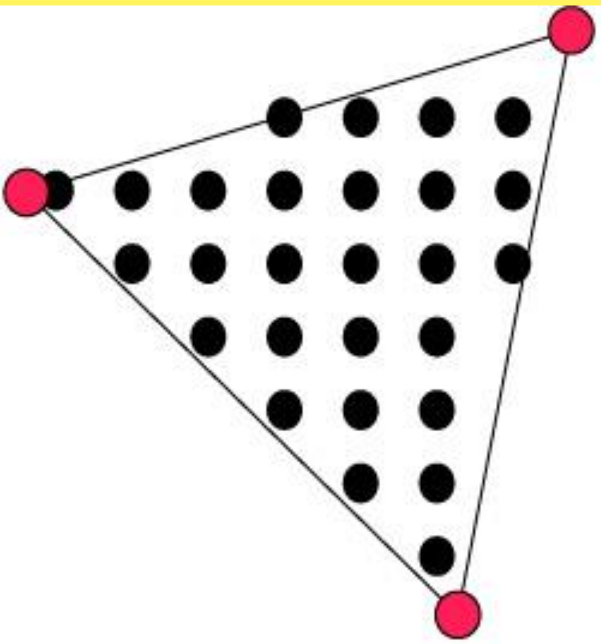
HARDWARE

Vertex shader → Geometry shader → Clipping → Screen mapping → Triangle Setup → Triangle Traversal → Pixel shader → Merger →



Display

What is vertex and fragment (pixel) shaders?

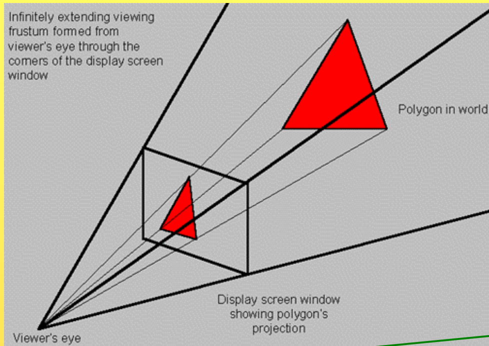


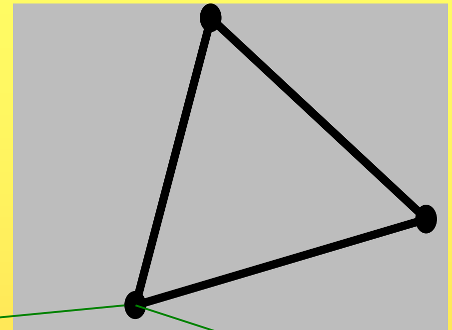
- Vertex shader: reads from textures
- Fragment shader: reads from textures, writes to pixel color
- Memory: Texture memory (read + write) typically 500 Mb – 4 GB
- Program size: the smaller the faster
- Instructions: mul, rcp, mov,dp, rsq, exp, log, cmp, jnz…

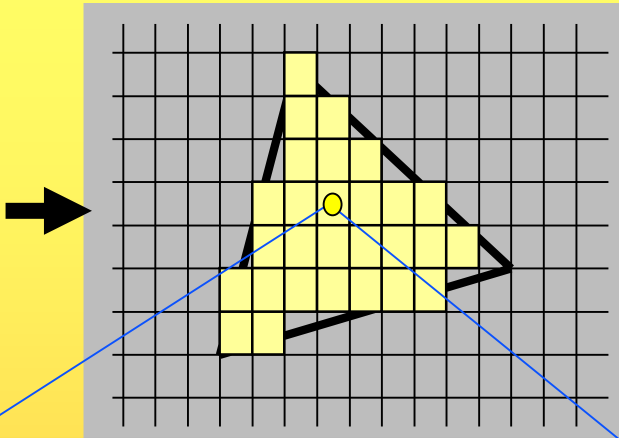
For each vertex, a vertex program (vertex shader) is executed

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

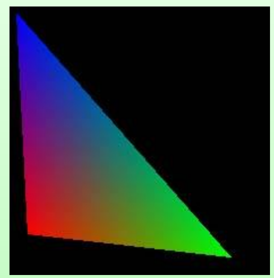
Shaders







```
// Vertex Shader
#version 130

in  vec3 vertex;
in  vec3 color;
out vec3 outColor;
uniform mat4 modelViewProjectionMatrix;

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

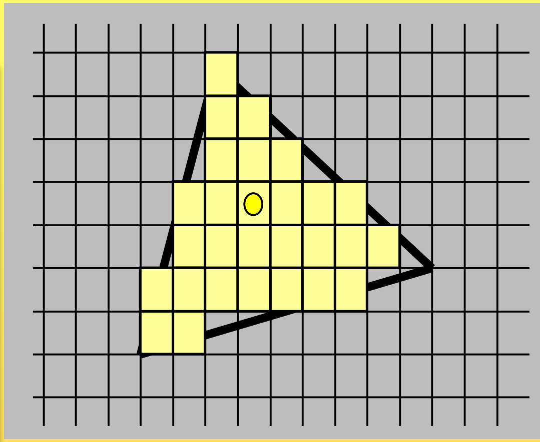


```
// Fragment Shader:
#version 130
in  vec3 outColor;
out vec4 fragColor;

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

Fragment Shader

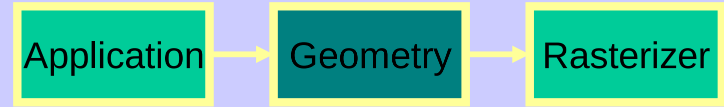
```
precision highp float;

varying vec2 uv_0;
varying vec3 n;

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

uniform float a;

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;
}
```

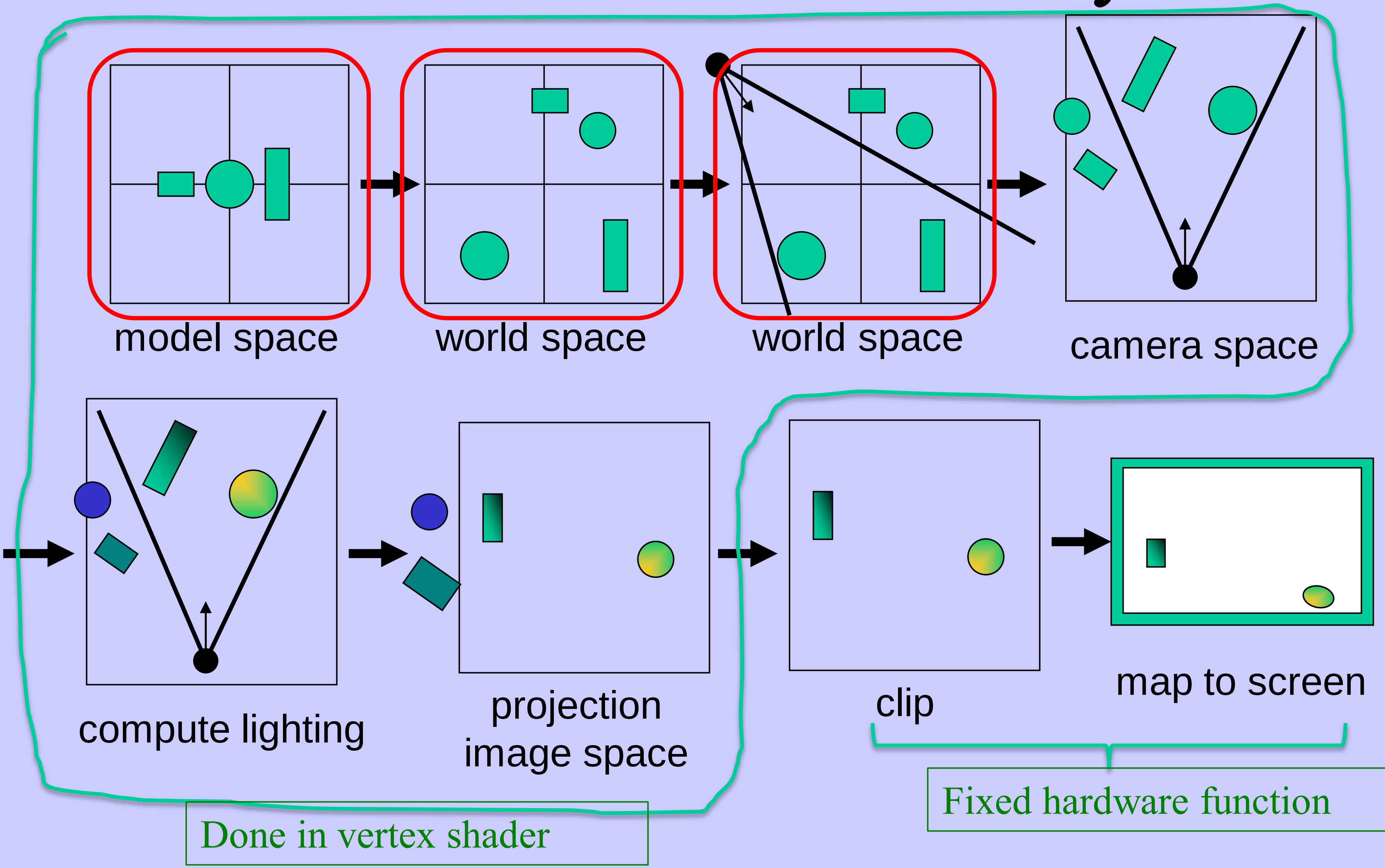


Rewind!
Let’s take a closer look

- 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

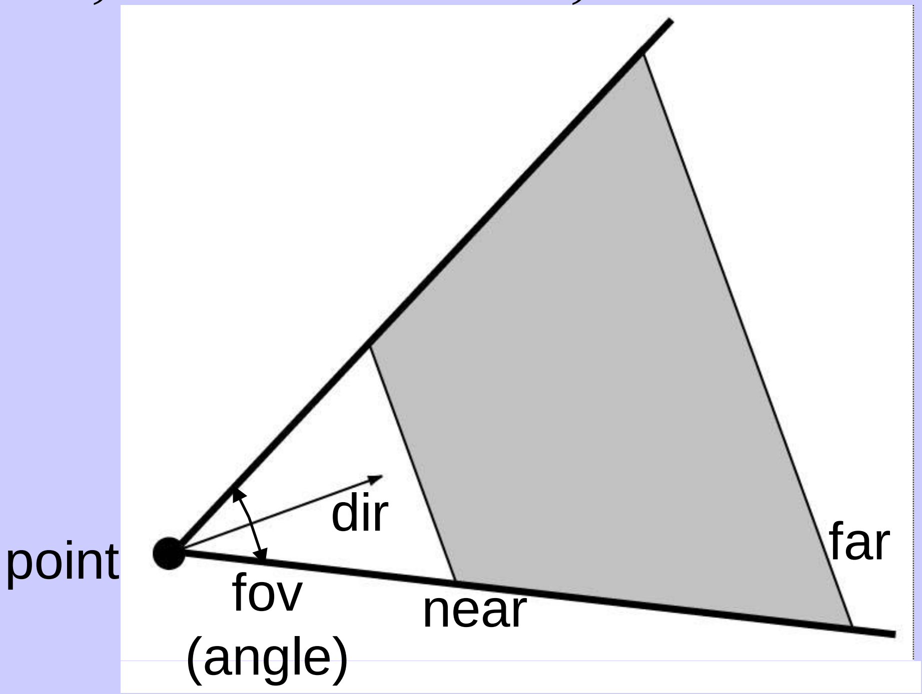


GEOMETRY - Summary



Virtual Camera

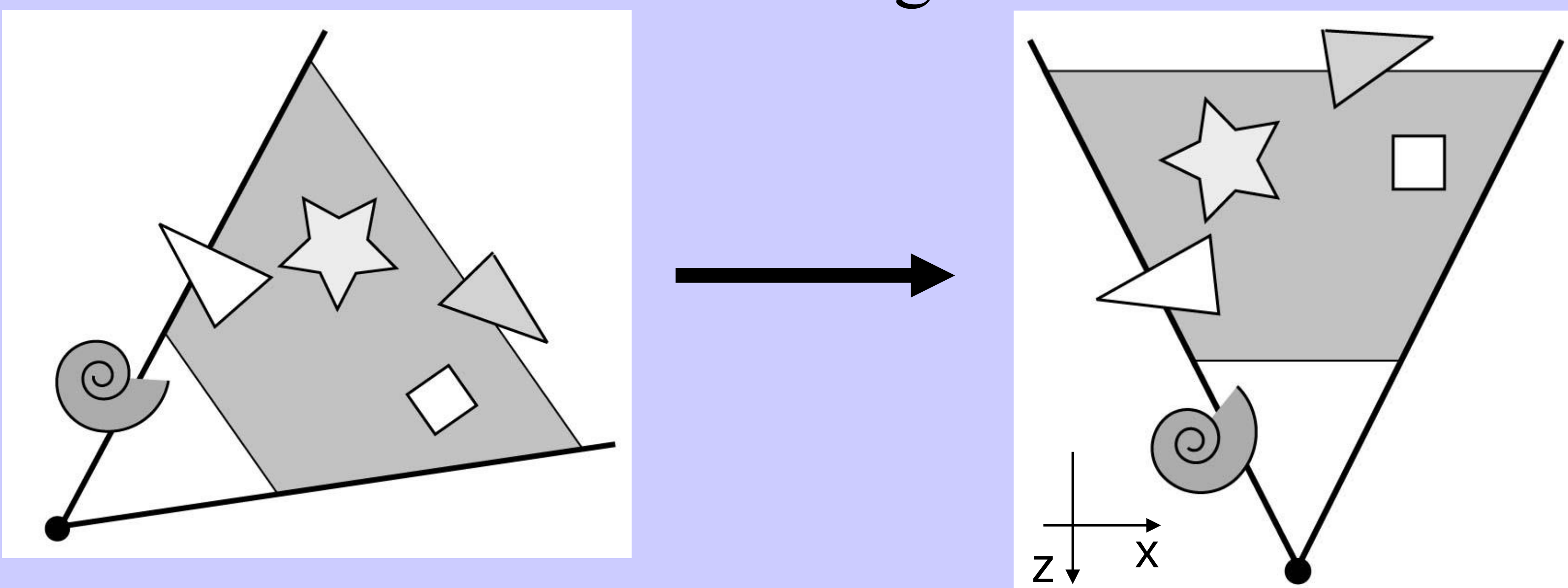
- Defined by position, direction vector, up vector, field of view, near and far plane.



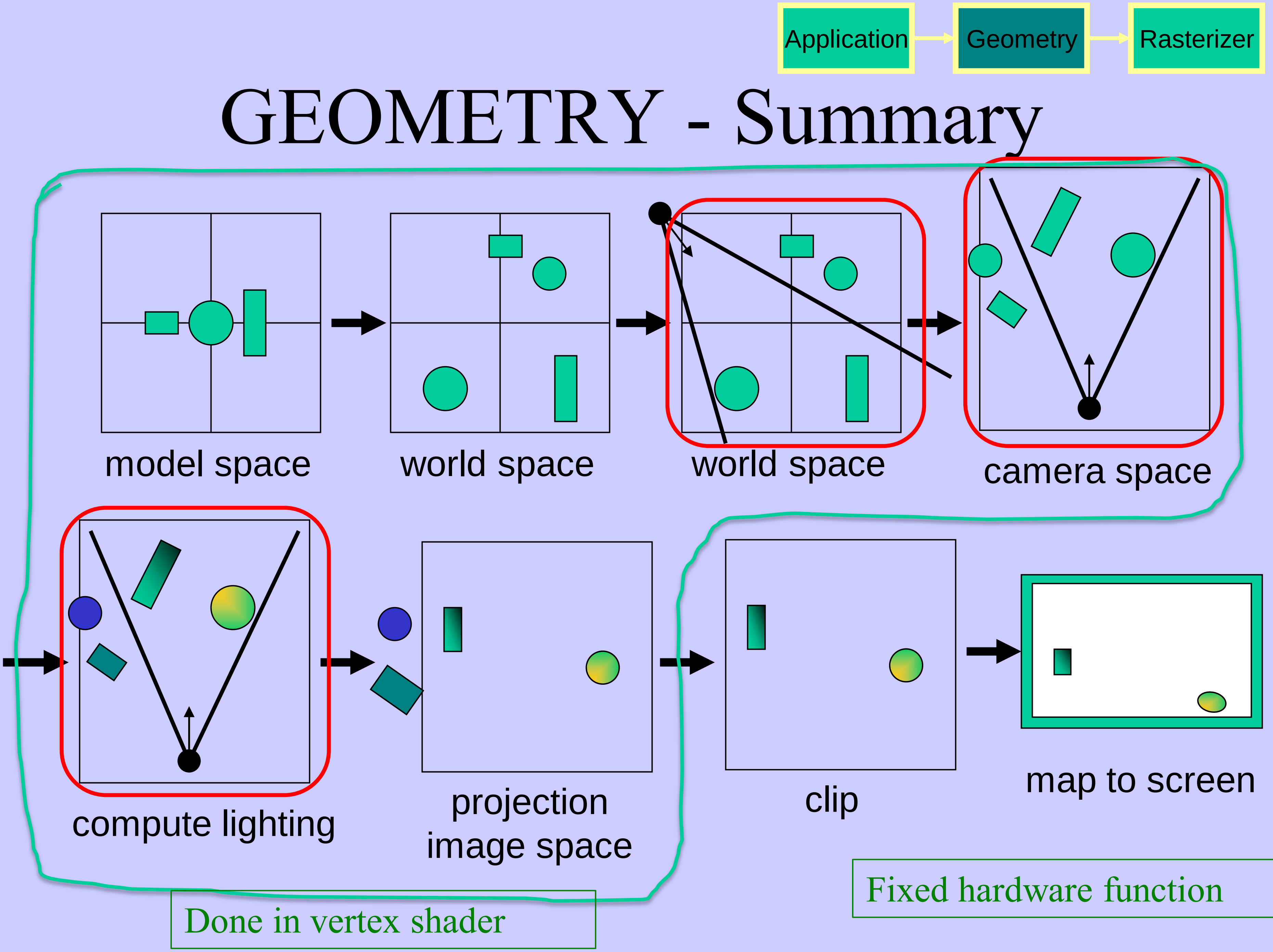
- Create image of geometry inside gray region
- Used by OpenGL, DirectX, ray tracing, etc.

GEOMETRY - The view transform

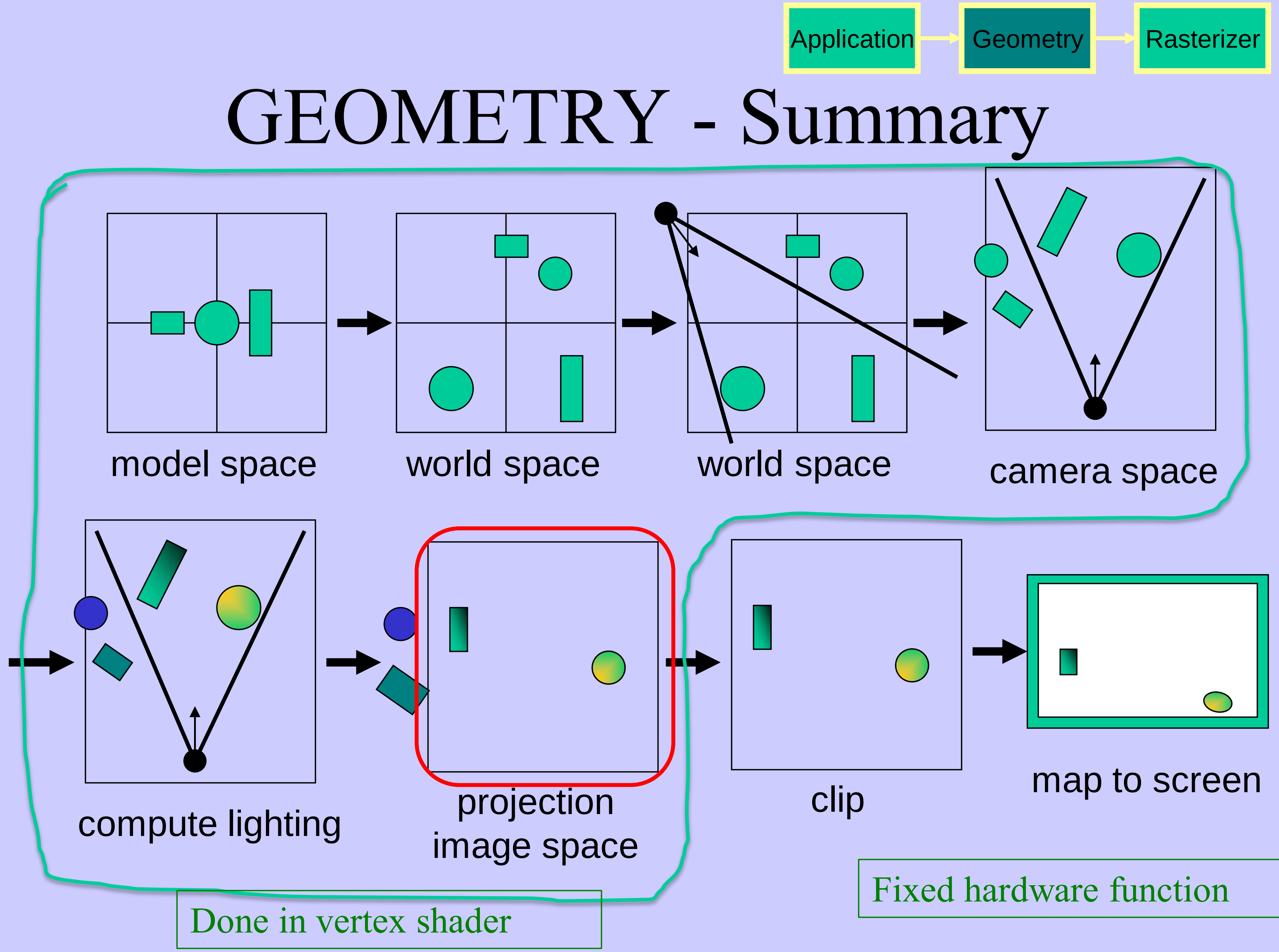
- You can move the camera in the same manner as objects
- But apply inverse transform to objects, so that camera looks down negative z-axis



GEOMETRY - Summary

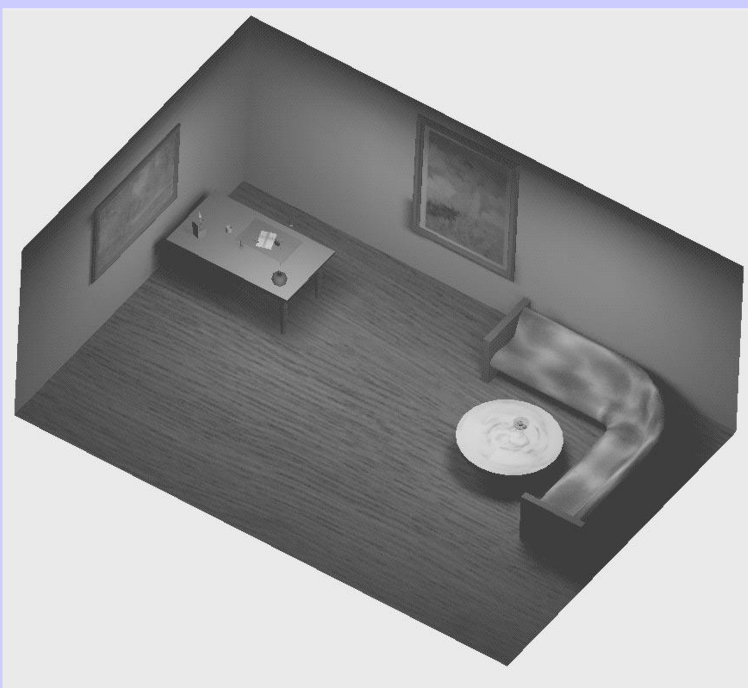


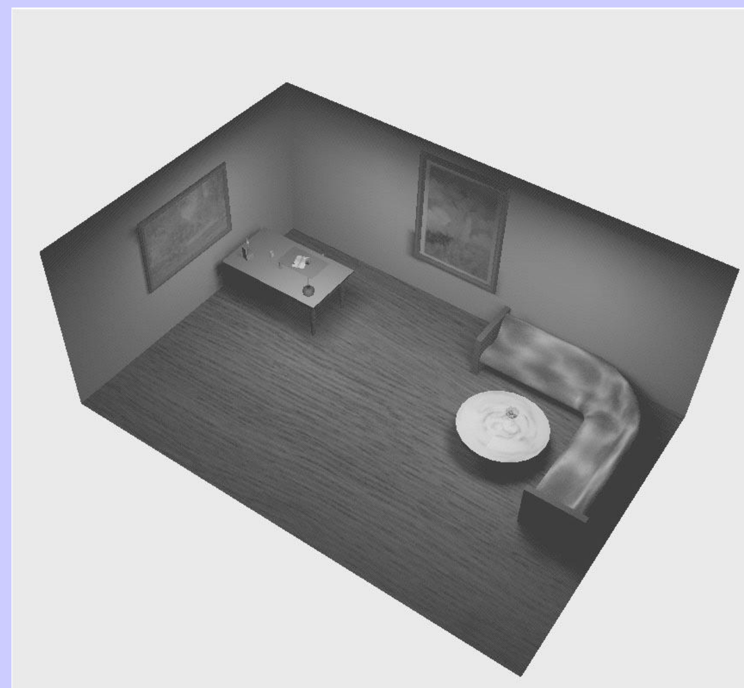
GEOMETRY - Summary



GEOMETRY - Projection

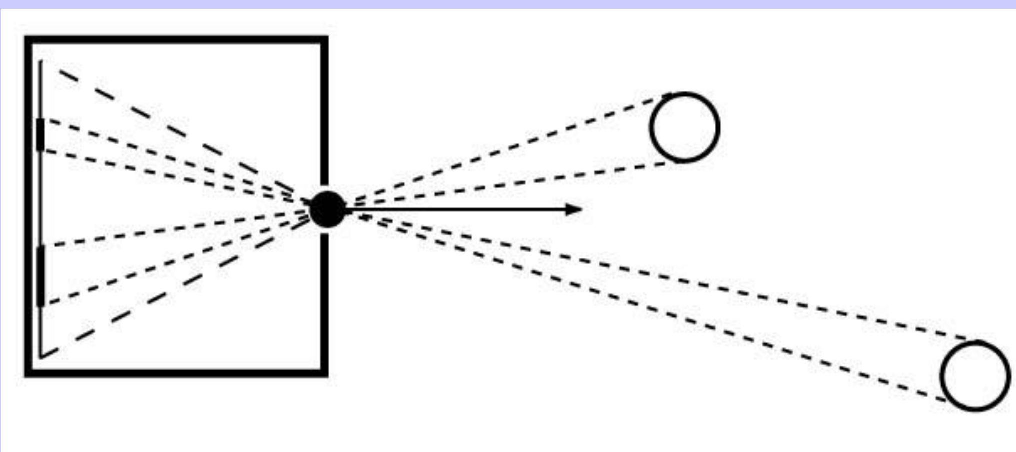
- Two major ways to do it
  - Orthogonal (useful in few applications)
  - Perspective (most often used)
    - Mimics how humans perceive the world, i.e., objects' apparent size decreases with distance

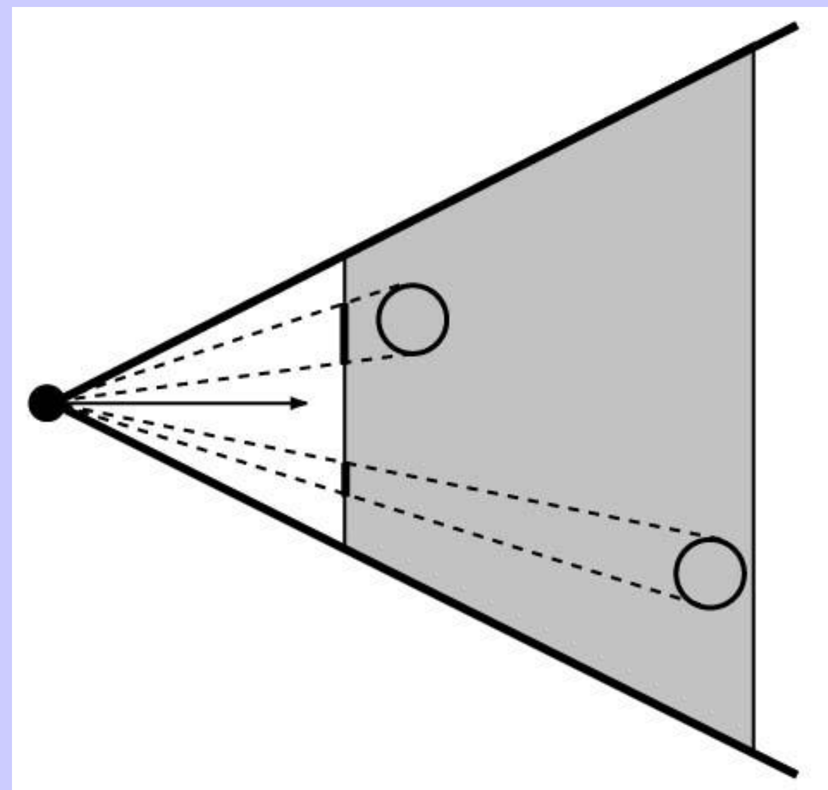




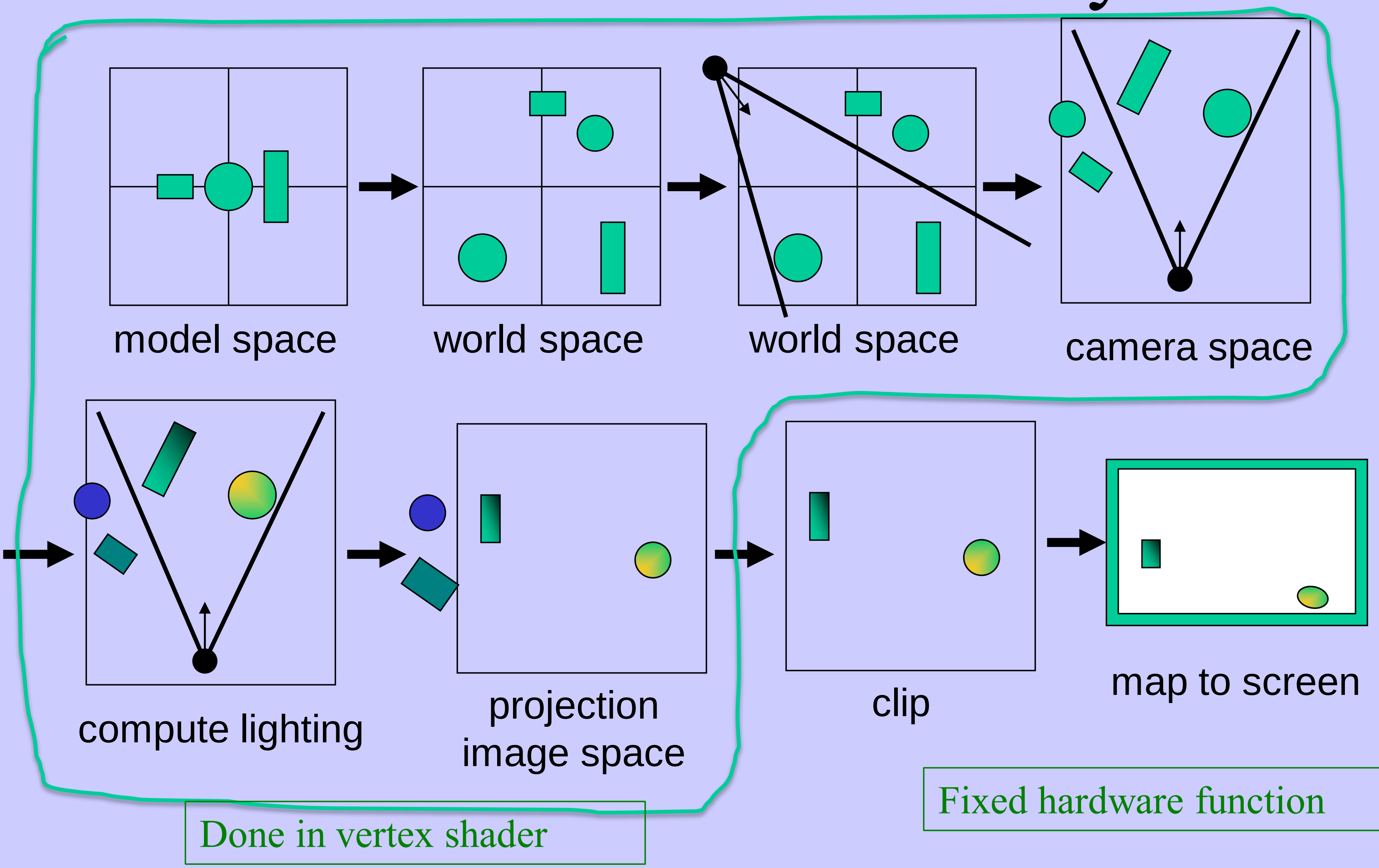
GEOMETRY - Projection

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





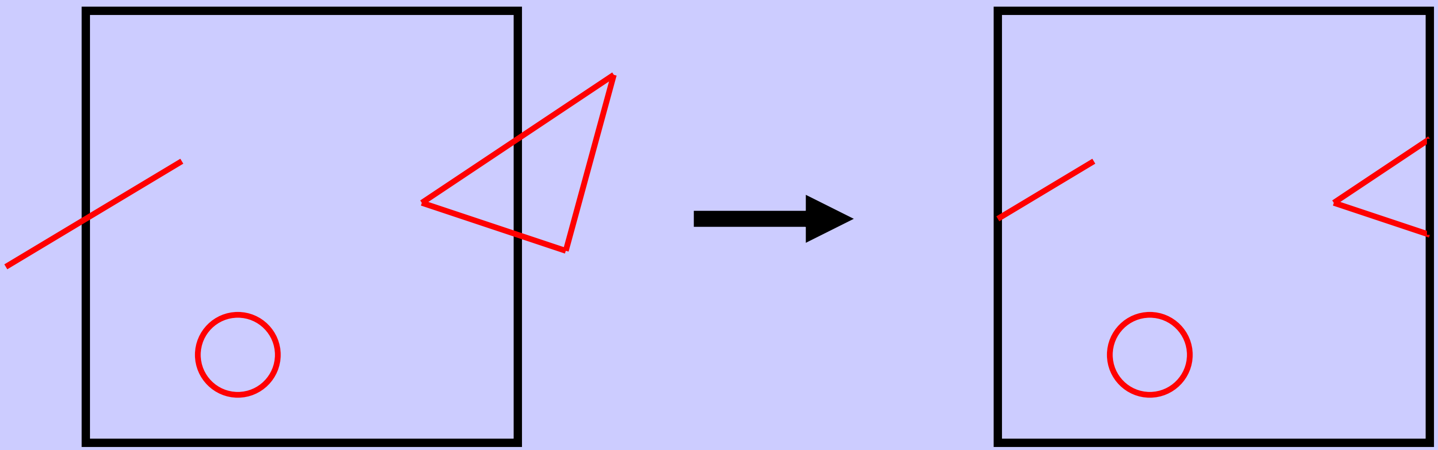
GEOMETRY - Summary



GEOMETRY

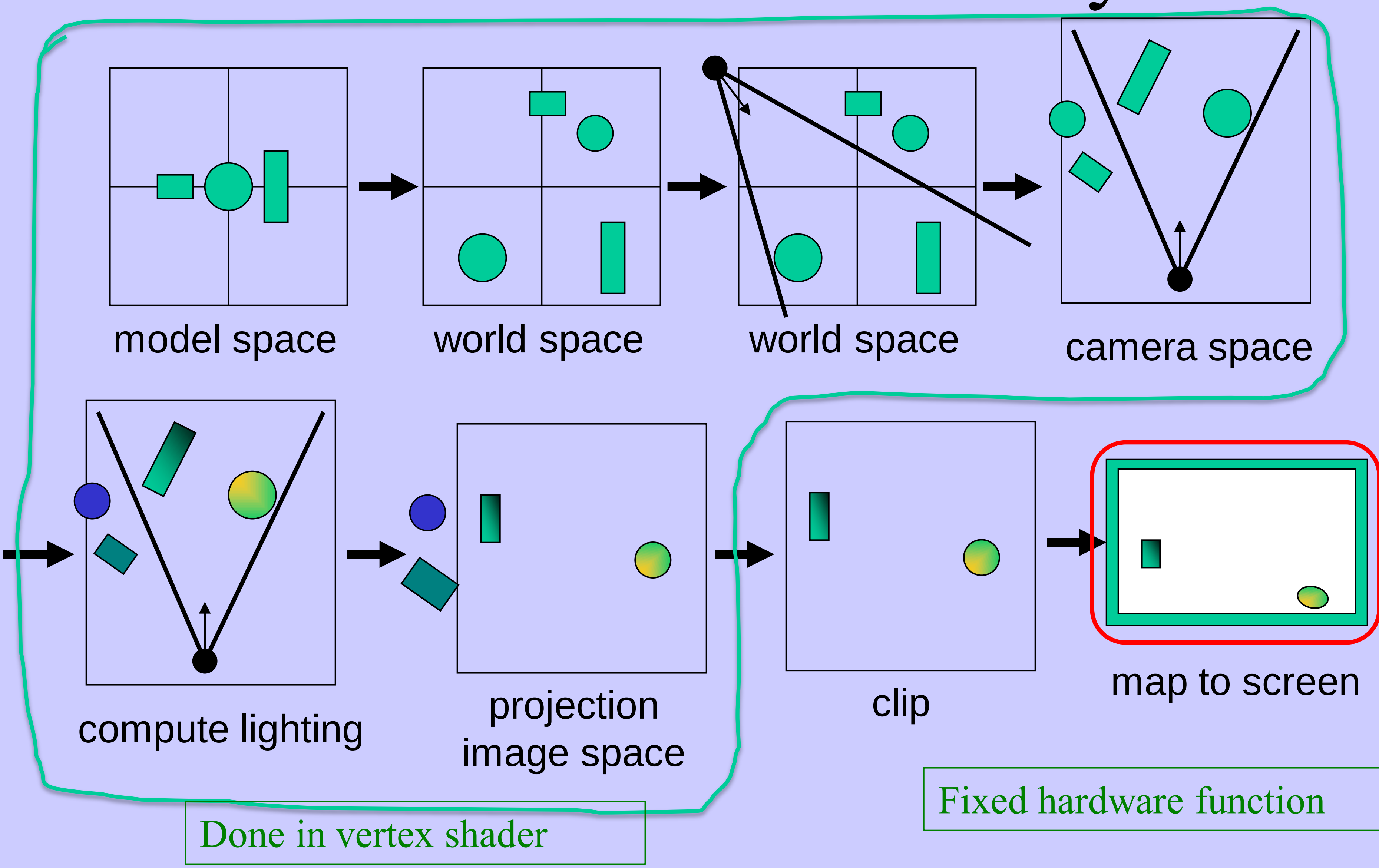
Clipping and Screen Mapping

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

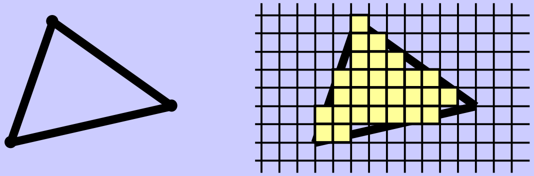


- Screen mapping, scales and translates the square so that it ends up in a rendering window
- These ”screen space coordinates” together with Z (depth) are sent to the rasterizer stage

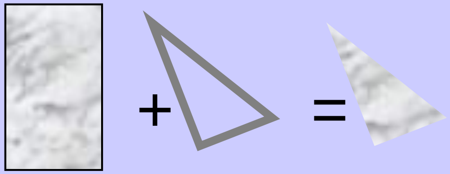
GEOMETRY - Summary

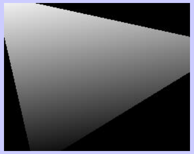


The RASTERIZER in more detail

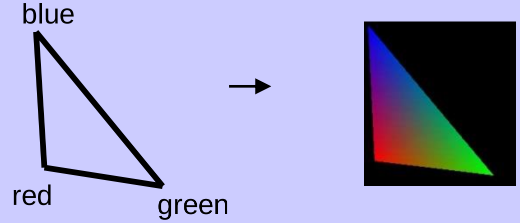


- Scan-conversion
  - Find out which pixels are inside the primitive
- Fragment shaders
  - E.g. put textures on triangles





  - Use interpolated data over triangle
  - and/or compute per-pixel lighting

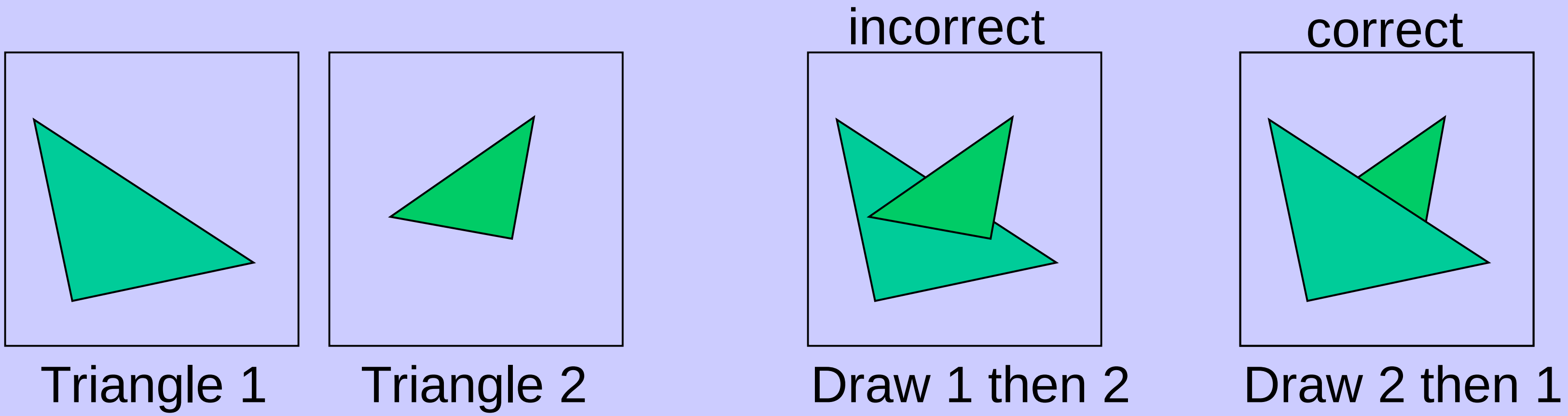


- Z-buffering
  - Make sure that what is visible from the camera really is displayed
- Doublebuffering

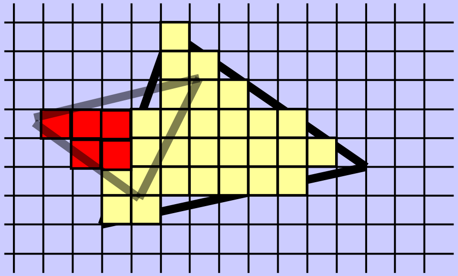
The RASTERIZER

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



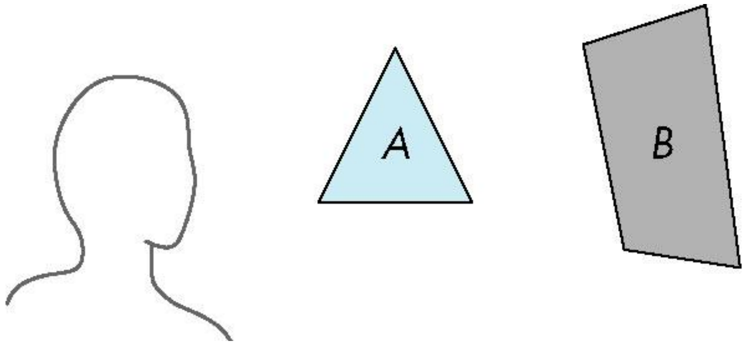
The RASTERIZER Z-buffering



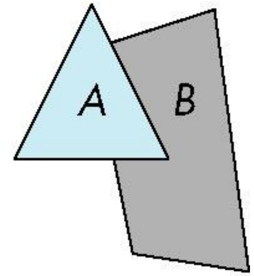
- Would be nice to avoid sorting…
- The Z-buffer (aka depth buffer) solves this
- Idea:
  - Store z (depth) at each pixel
  - When rasterizing a triangle, compute z at each pixel on triangle
  - Compare triangle’s z to Z-buffer z-value
  - If triangle’s z is smaller, then replace Z-buffer and color buffer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

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



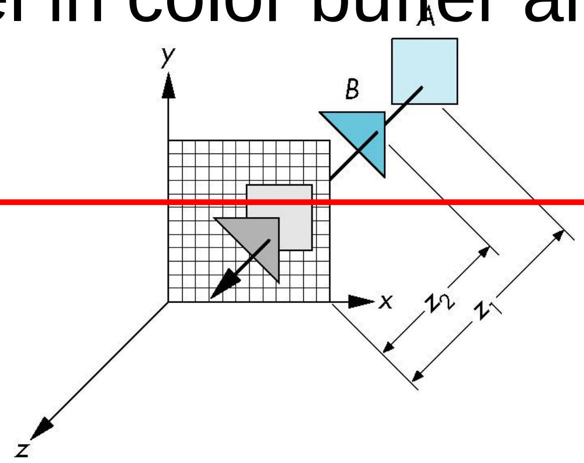
Fill B then A

- 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 pixel to depth in z buffer
- If less, place shade of pixel in color buffer and update z buffer



The RASTERIZER double-buffering

- 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

Screen Tearing

Swapping
back/front buffers

vblank

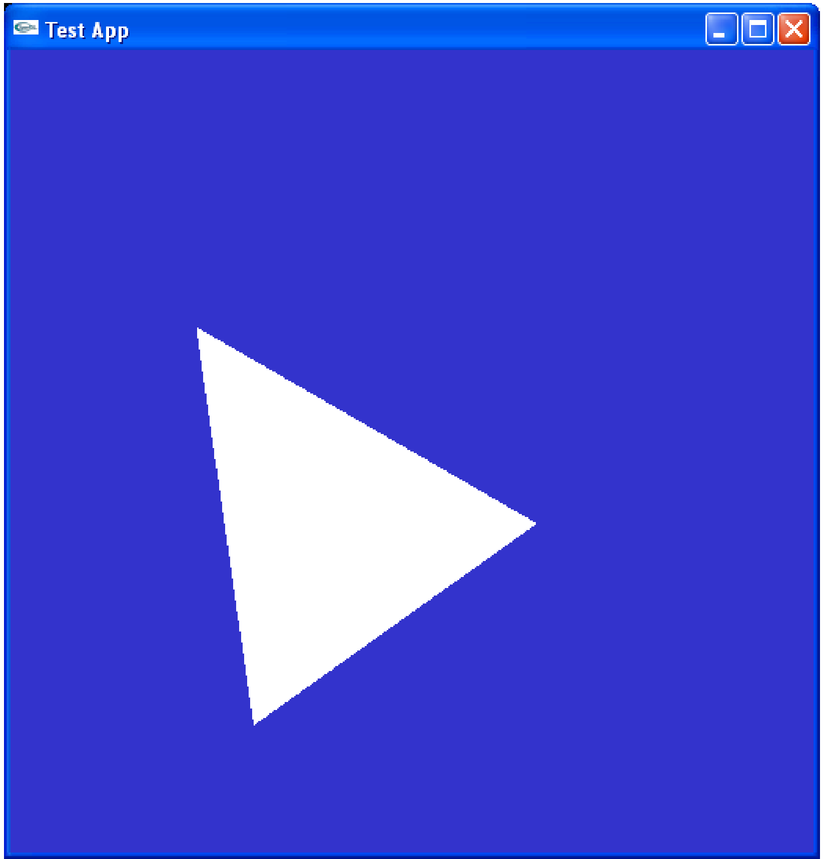
Screen Tearing

- 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.

OpenGL

A Simple Program
Computer Graphics version of "Hello World"

Generate a triangle on a solid background

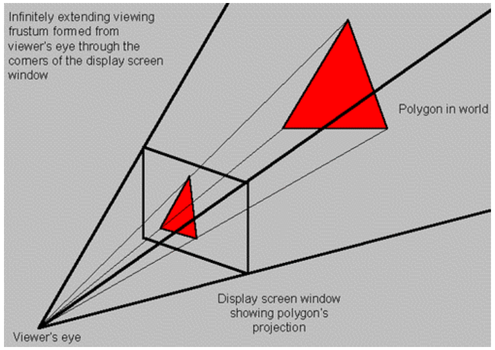


Simple Application...

```
int main(int argc, char *argv[])
{
    glutInit(&argc, argv);

    /* open window of size 512x512 with double buffering, RGB colors, and Z-
    buffering */
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(512,512);
    glutCreateWindow("Test App");

    /* the display function is called once when the gluMainLoop is called,
     * but also each time the window has to be redrawn due to window
     * changes (overlap, resize, etc).  */
    glutDisplayFunc(display);        // Set the main redraw function


    glutMainLoop();  /* start the program main loop */
    return 0;
}
```

```
void display(void)
{
	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

	int w = glutGet((GLenum)GLUT_WINDOW_WIDTH);
	int h = glutGet((GLenum)GLUT_WINDOW_HEIGHT);
	glViewport(0, 0, w, h);				// Set viewport (OpenGL draws with this resolution)

	glDisable(GL_CULL_FACE);
	drawScene();

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

```
static void drawScene(void)
{
	// Shader Program
	glUseProgramObjectARB( shaderProgram );  // Set the shader program to use for this draw call
	CHECK_GL_ERROR();

	glBindVertexArray(vertexArrayObject);          // Tells which vertex arrays to use
	CHECK_GL_ERROR();

	glDrawArrays( GL_TRIANGLES, 0, 3 );        // Render the three first vertices as a triangle
	CHECK_GL_ERROR();
}
```



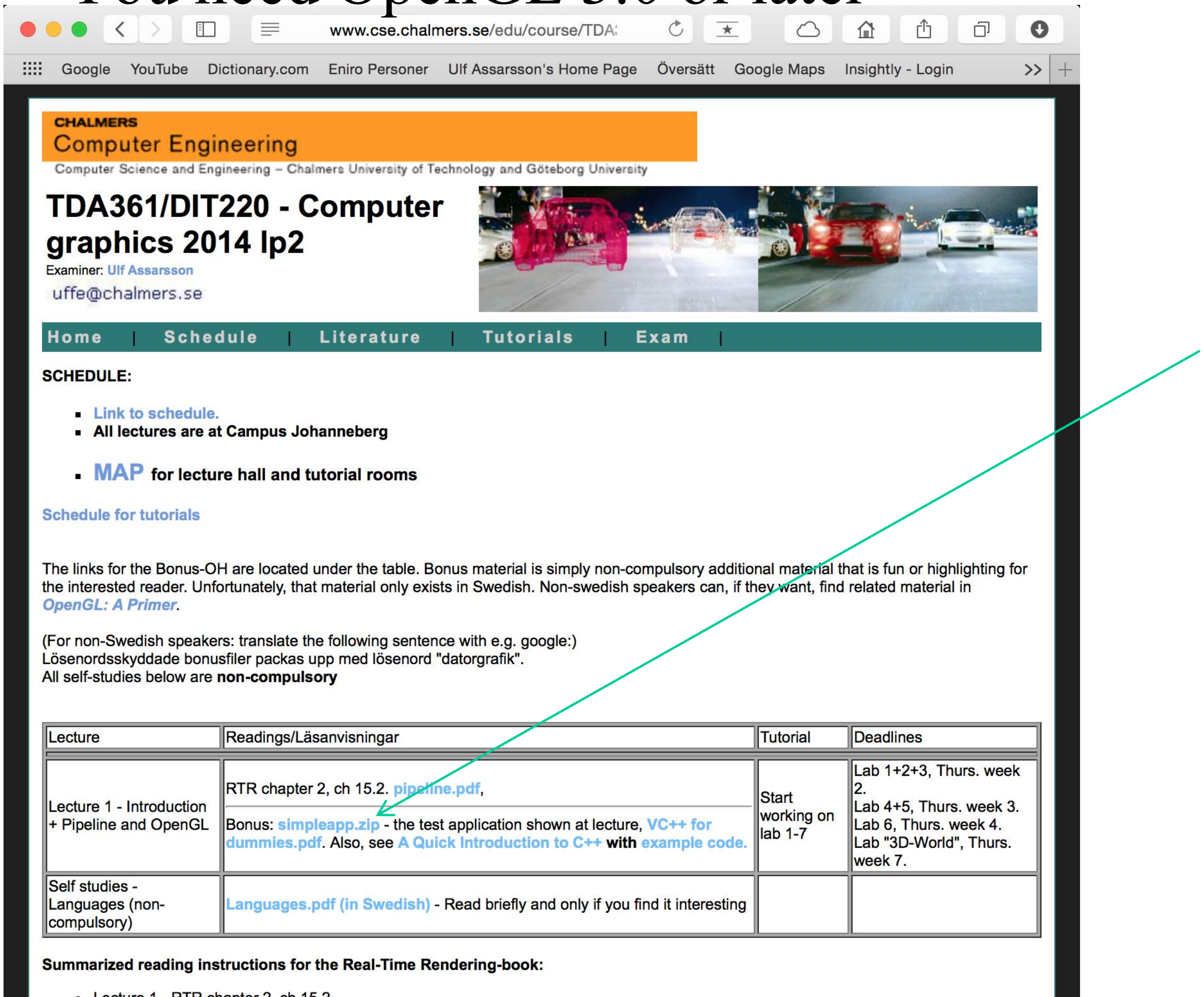
Shaders

```
// Vertex Shader
#version 130

in  vec3 vertex;
in  vec3 color;
out vec3 outColor;
uniform mat4 modelViewProjectionMatrix;

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

```
// Fragment Shader:
#version 130
in  vec3 outColor;
out vec4 fragColor;

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

Demonstration of SimpleApp

- Available on course homepage in Schedule.
- You need OpenGL 3.0 or later



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

TDA361/DIT220 - Computer graphics 2014 lp2
Examiner: Ulf Assarsson
uffe@chalmers.se

Home | Schedule | Literature | Tutorials | Exam |

SCHEDULE:

- Link to schedule.
- **All lectures are at Campus Johanneberg**
- MAP **for lecture hall and tutorial rooms**

Schedule for tutorials

The links for the Bonus-OH are located under the table. Bonus material is simply non-compulsory additional material that is fun or highlighting for the interested reader. Unfortunately, that material only exists in Swedish. Non-swedish speakers can, if they want, find related material in *OpenGL: A Primer*.

(For non-Swedish speakers: translate the following sentence with e.g. google:)
Lösenordsskyddade bonusfiler packas upp med lösenord "datorgrafik".
All self-studies below are **non-compulsory**

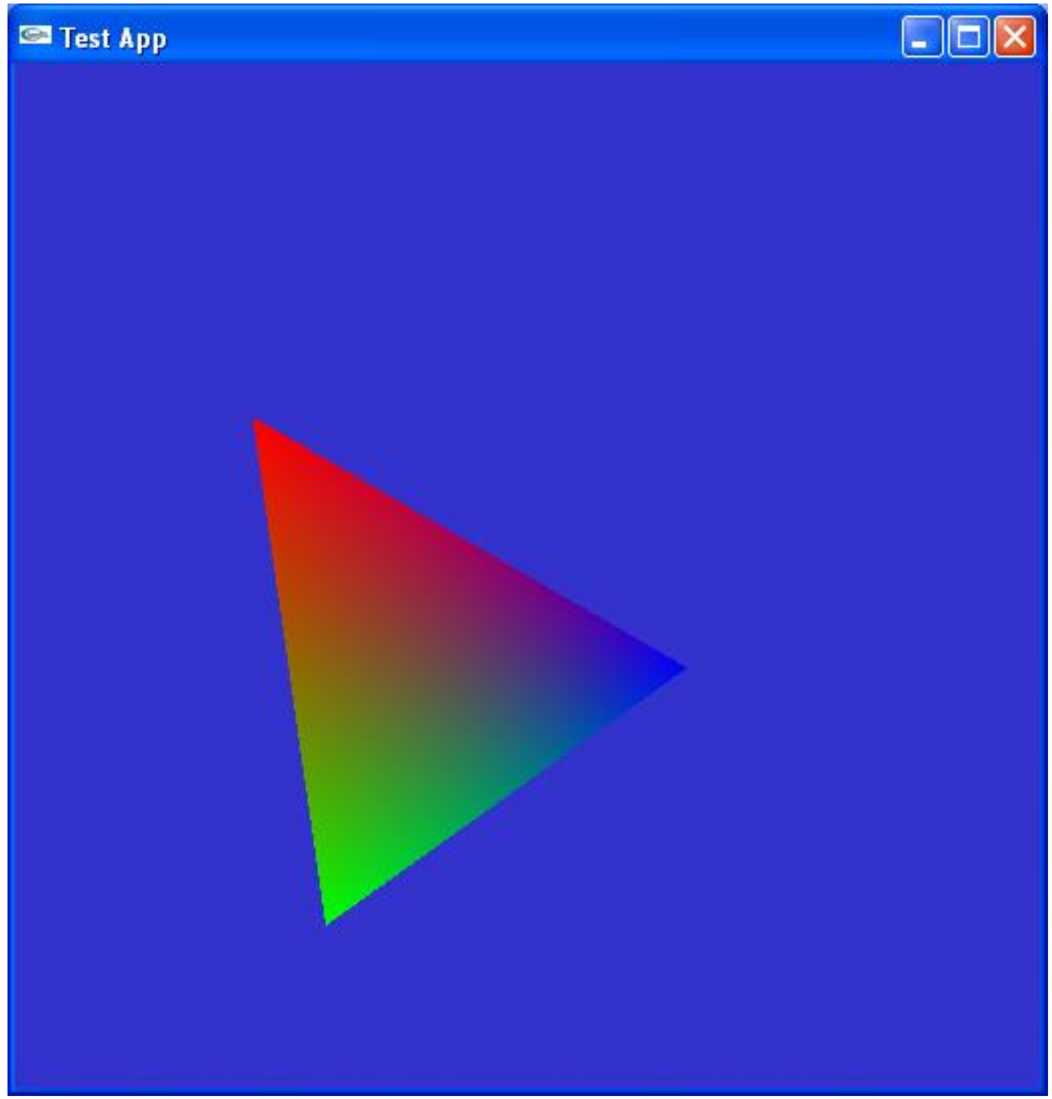
Lecture	Readings/Läsanvisningar	Tutorial	Deadlines
Lecture 1 - Introduction + Pipeline and OpenGL	RTR chapter 2, ch 15.2. pipeline.pdf, Bonus: simpleapp.zip - the test application shown at lecture, VC++ for dummies.pdf. Also, see A Quick Introduction to C++ **with** example code.	Start working on lab 1-7	Lab 1+2+3, Thurs. week 2. Lab 4+5, Thurs. week 3. Lab 6, Thurs. week 4. Lab "3D-World", Thurs. week 7.
Self studies - Languages (non-compulsory)	Languages.pdf (in Swedish) - Read briefly and only if you find it interesting		

Summarized reading instructions for the Real-Time Rendering-book:

- Lecture 1 - RTR chapter 2, ch 15.2

Cool application

Starts looking good!



Repetition

- What is important:
  - Understand the Application-, Geometry- and Rasterization Stage

- See you on Friday 9:00