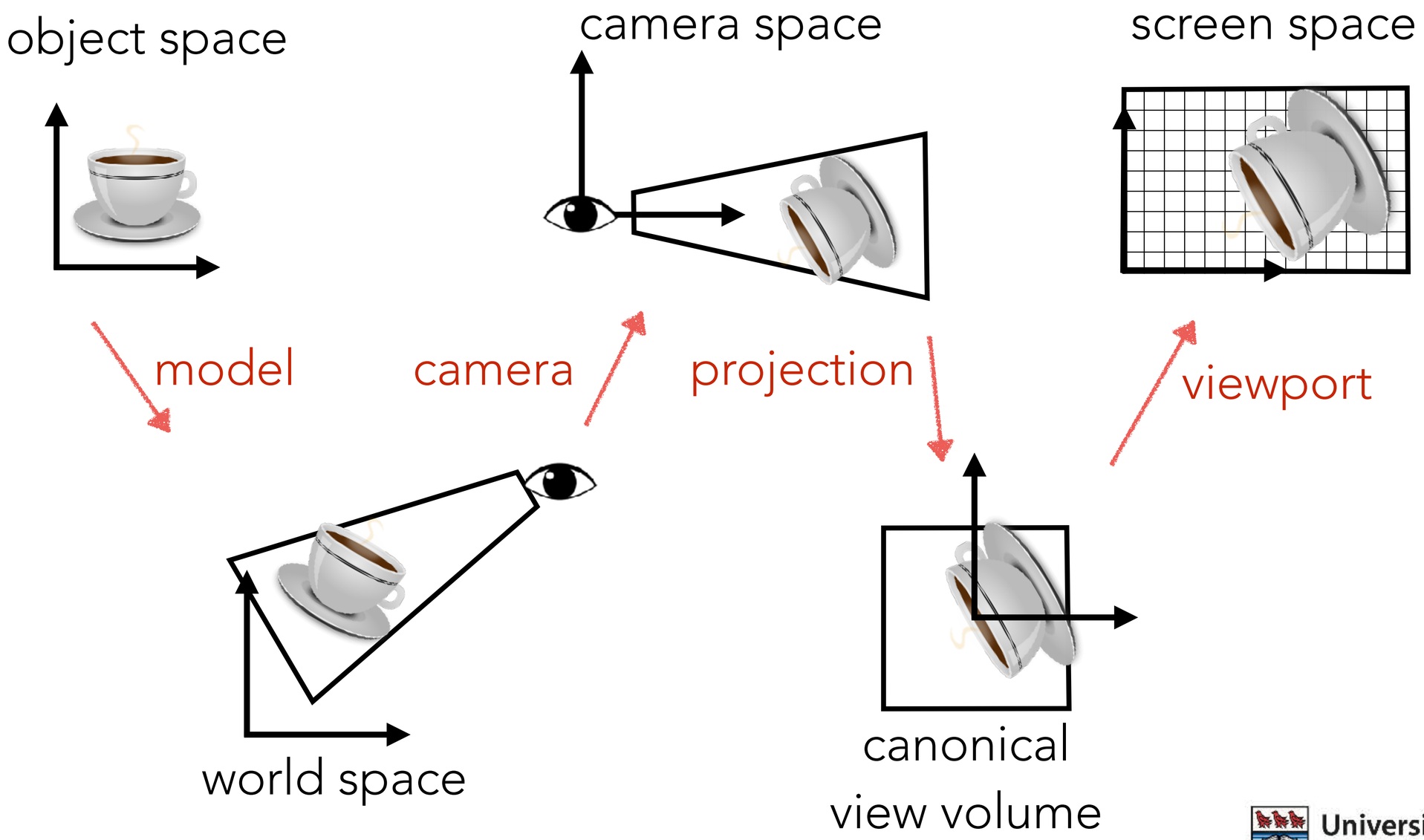
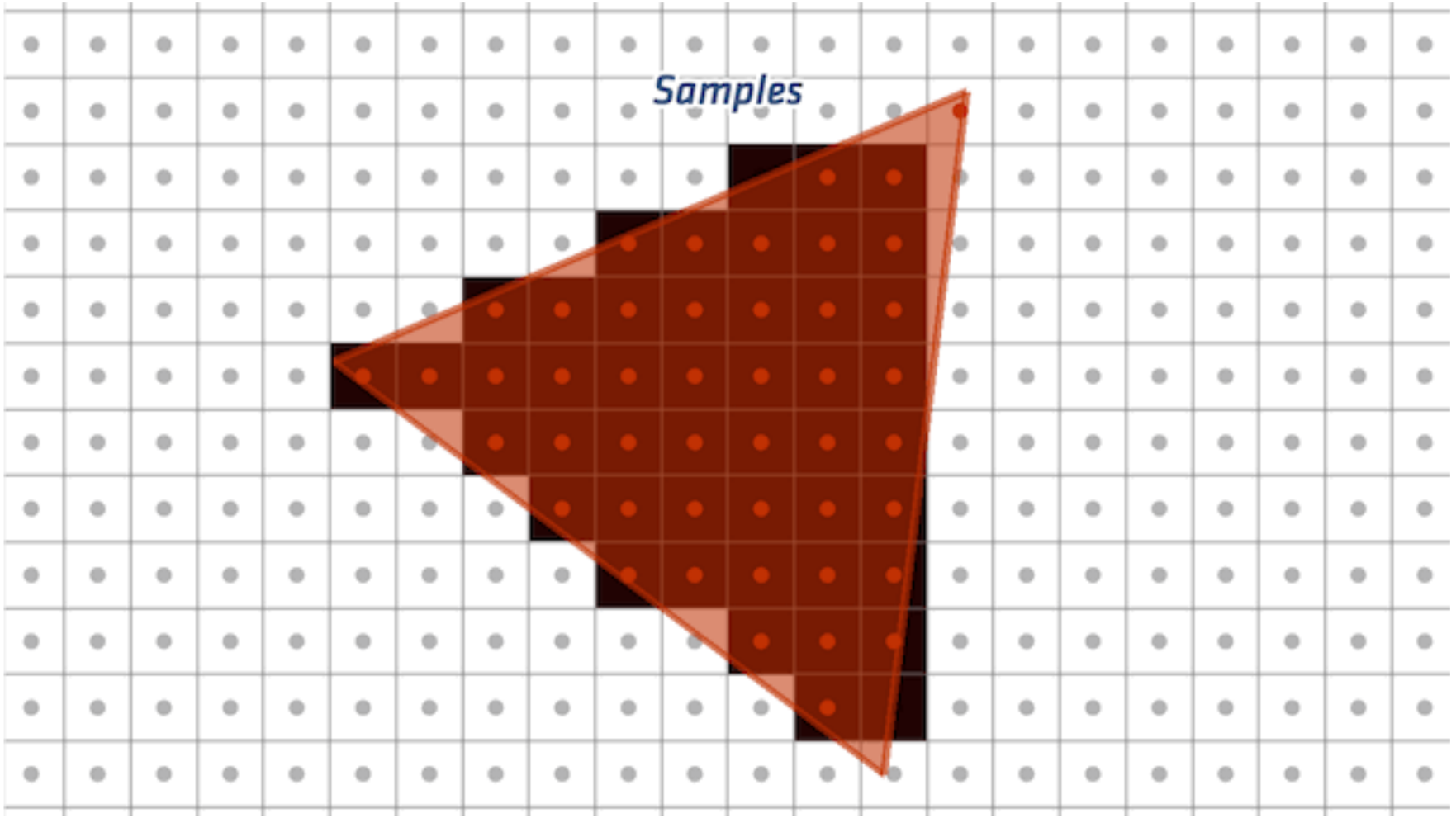
Rasterization - Implementation

Recap: Viewing Transformation





How to do it?

- Specialized hardware — Graphics Processing Unit (GPU)
- APIs to interact with the hardware
  - OpenGL/Vulkan (Windows, Linux, Android)
  - DirectX (Windows)
  - Metal (MacOS X, iOS)

Context Creation

- Before you can draw anything you need to:
  - Open a window (i.e. ask the OS to give you access to a portion of the screen)
  - Initialize the API and assign the available screen space to it
- This is a technical step and it is heavily dependent on the operating system and on the hardware

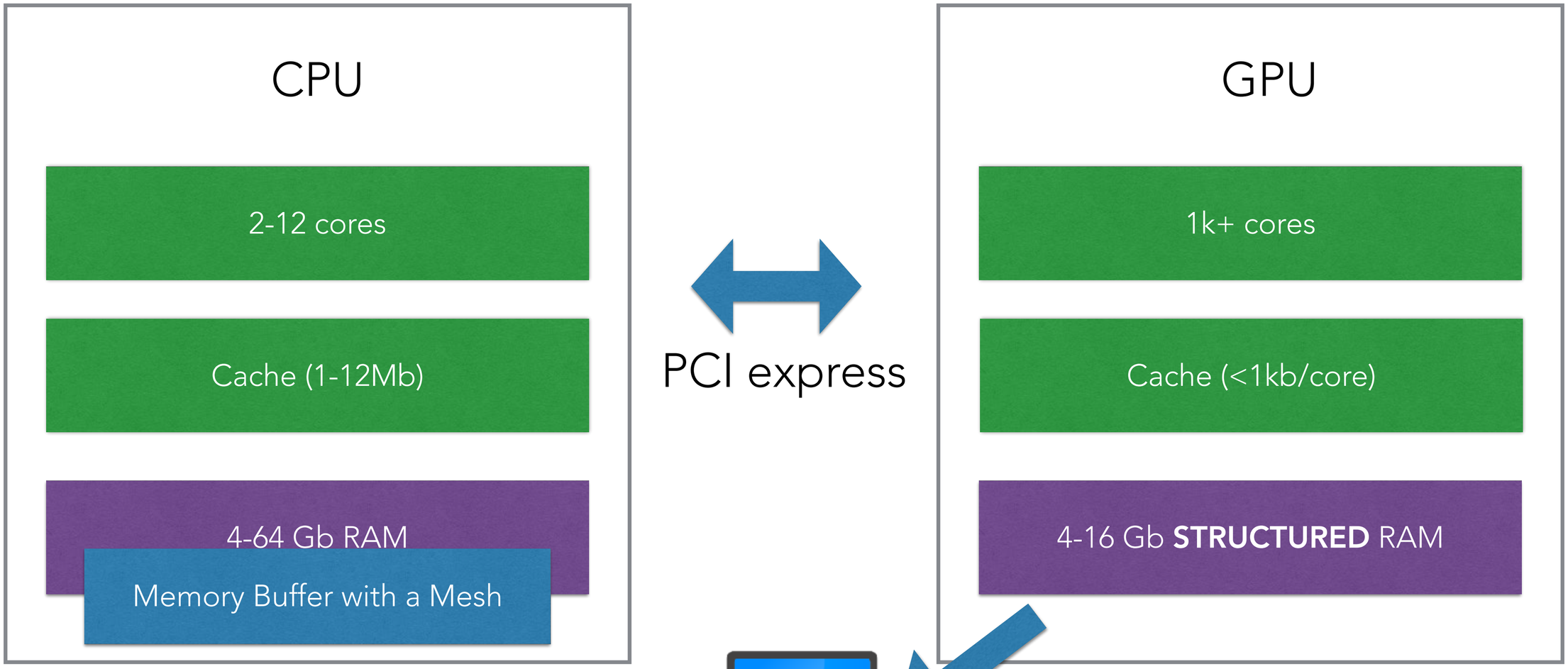
Window Manager

- There are many libraries that take care of this for you, hiding all the complexity and providing a cross-platform and cross-hardware interface
- Some examples are GLFW and SDL
- A window manager usually provides an event management system

CPU and GPU memory

You write code here …

… which acts here.



Technical and OS-Specific

- Writing code for the GPU makes debugging harder, since the standard debugging tools cannot be used
- The code needs to be customized for the specific operating system you are developing for
- The good news is that all modern APIs, at a high-level, offer the same concepts and the same features
- We will thus study a toy software rasterizer that, while slower than hardware solutions, will allow you to fully understand how rasterization works. The jump from the software rasterizer to any modern API is small and mostly technical

Software Rasterization

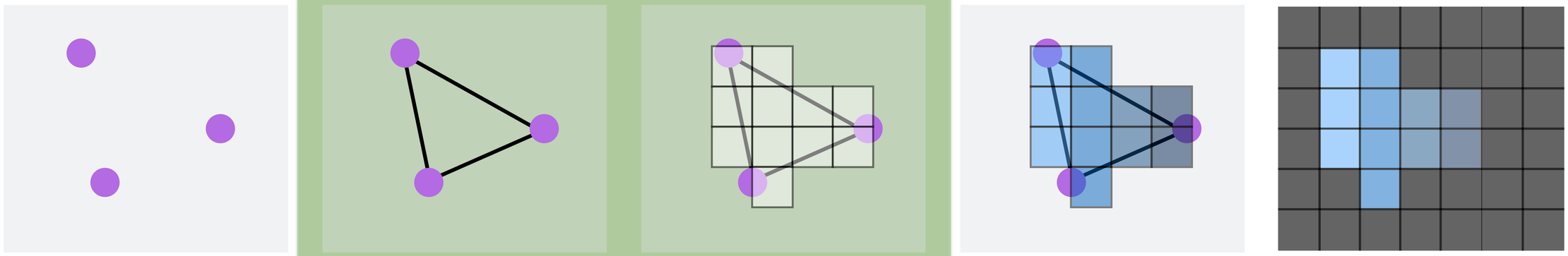
raster.h/raster.cpp

- The rasterizer mimics the API of modern OpenGL4/Vulkan/DirectX12
- It is minimalistic, supporting only rendering of triangles and lines
- It is readable, less than 200 lines of C++ code
- It is expected that you understand every single line after this slideset
- It allows to render the same scenes as HW solutions, it is just slower
- I recommend to render small scenes in debug mode, and then switch to release for real ones

Rasterization Pipeline

Input
Vertex Attributes
Uniform Attributes

Output
RGBA Image



Vertex Input

- You have to send to the rasterizer a set of vertex attributes:
  - World Coordinates
  - Color
  - Normal
- You can pass as many as you want, but remember that memory/ bandwidth is precious, you want to send only what is required by the shaders

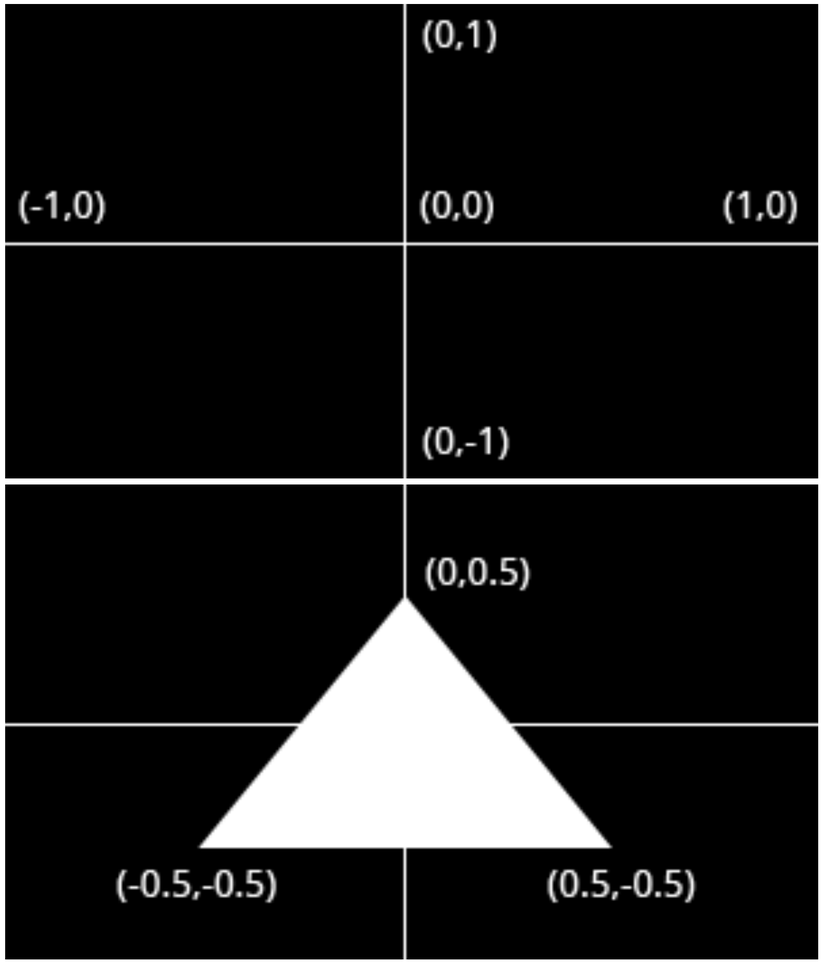
Drawing a Triangle

- Only the triangles in the canonical cube will be rendered. The cube will be stretched to fill all available screen space.

```
vector<VertexAttributes> vertices;
vertices.push_back(VertexAttributes(-0.5,-0,5));
vertices.push_back(VertexAttributes(0.5,-0.5,0));
vertices.push_back(VertexAttributes(0,0.5,0));

rasterize_triangles(program,uniform,vertices,frameBuffer);

vector<uint8_t> image;
framebuffer_to_uint8(frameBuffer,image);
stbi_write_png("triangle.png", frameBuffer.rows(), frameBuffer.cols(), 4,
image.data(), frameBuffer.rows()*4);
```



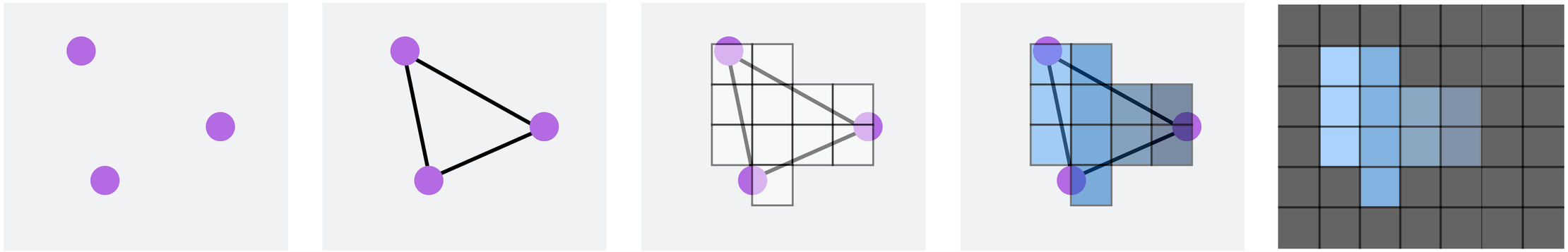
Shaders

- The name is historical — they were introduced to allow customization of the shading step of the traditional graphics pipeline
- Shaders are **general purpose functions** that will be executed in parallel on the GPU (serially in our software rasterizer, making it parallel is an interesting final project)
- They are usually written in a custom programming language, but in our case they will be standard C++ functions
- They allow to customize the behavior of the rasterizer to achieve a variety of effects

Rasterization Pipeline

Input
Vertex Attributes
Uniform Attributes

Output
RGBA Image



Vertex Shader

Primitive Assembly

Rasterization

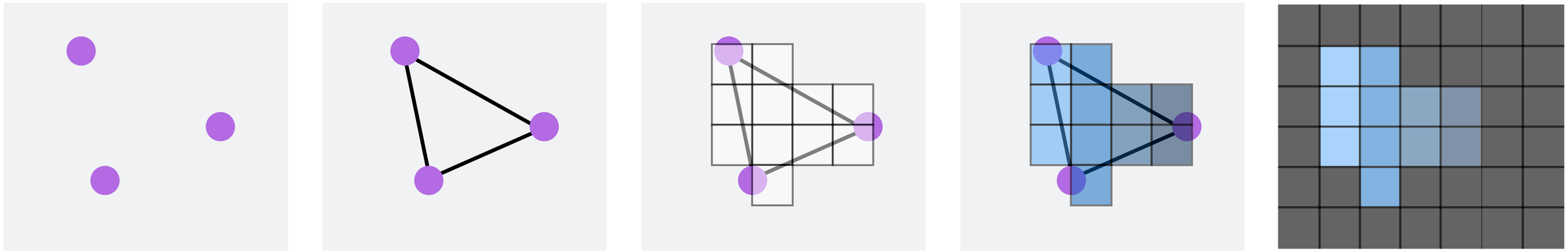
Fragment Shader

Blending Shader

Rasterization Pipeline

Input
Vertex Attributes
Uniform Attributes

Output
RGBA Image



Vertex Shader

Primitive Assembly

Rasterization

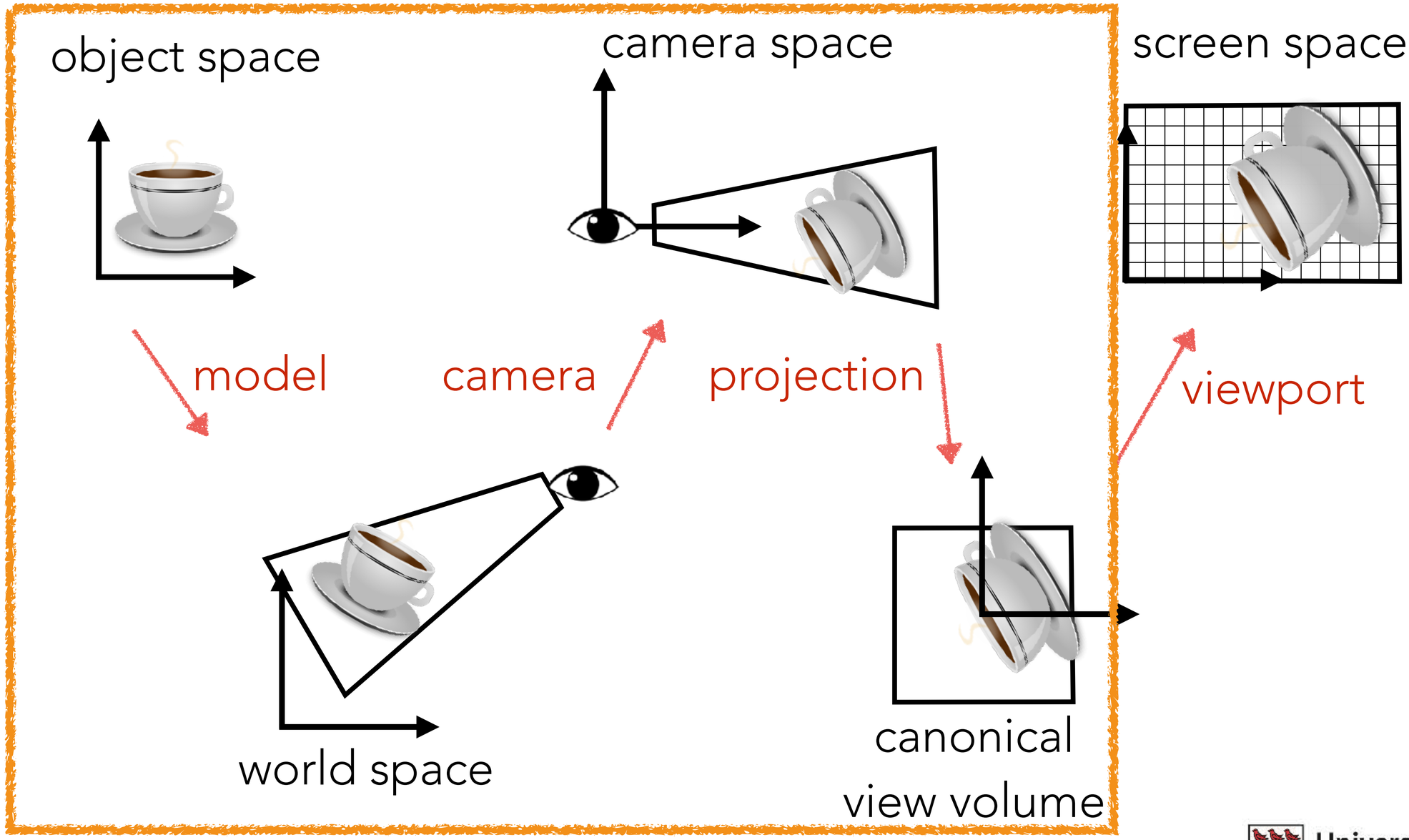
Fragment Shader

Blending Shader

Vertex Shader

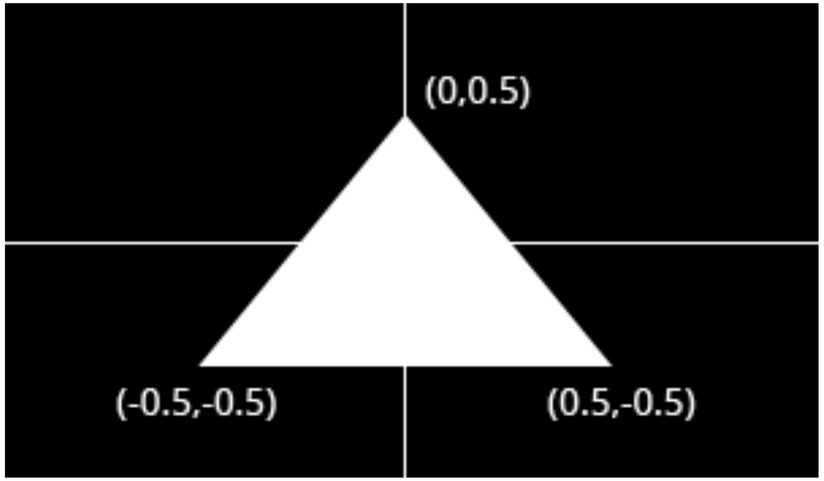
- The vertex shader is a function processing each vertex (and its attributes) as they appear in the input
- Its duty is to output the final vertex position in the canonical bi-unit cube and to output any data required by the fragment shader
- All transformations from **world to device coordinates** happen here

Vertex Shader



A Simple Vertex Shader

```
vector<VertexAttributes> vertices;
vertices.push_back(VertexAttributes(-0.5,-0,5));
vertices.push_back(VertexAttributes(0.5,-0.5,0));
vertices.push_back(VertexAttributes(0,0.5,0));
```



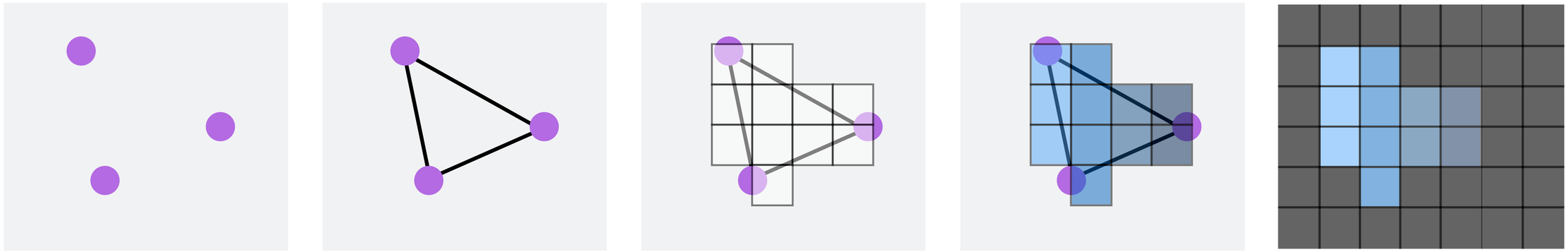

```
// The vertex shader is the identity
program.VertexShader = [](const VertexAttributes& va, const UniformAttributes& uniform)
{
    return va;
};
```

The syntax [] defines a lambda function.

Rasterization Pipeline

Input
Vertex Attributes
Uniform Attributes

Output
RGBA Image



Vertex Shader

Primitive Assembly

Rasterization

Fragment Shader

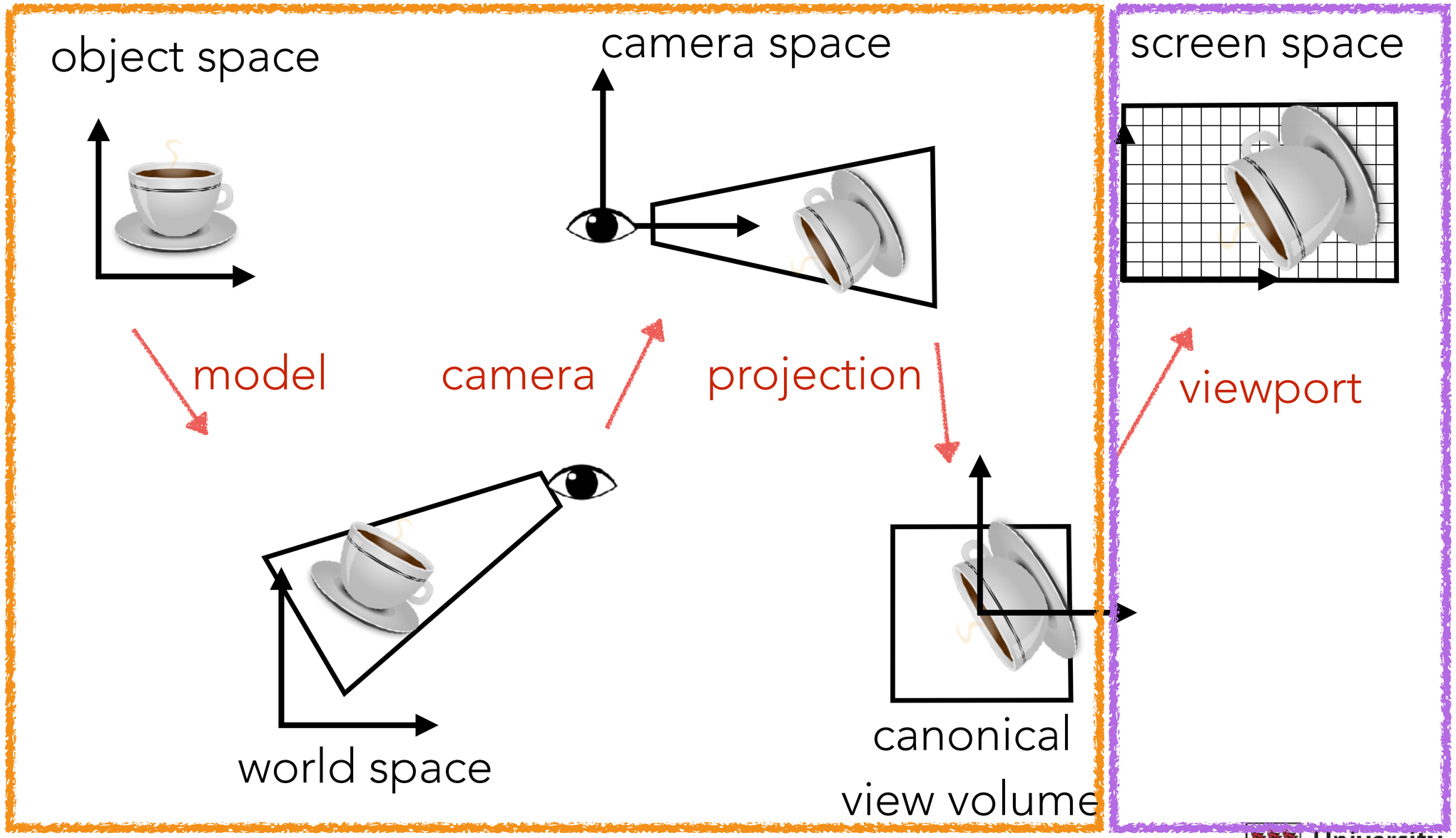
Blending Shader

Fragment Shader

- The output from the vertex shader is interpolated over all the pixels on the screen covered by a primitive
- These pixels are called fragments and this is what the fragment shader operates on
- It has one mandatory output, the final color of a fragment. It is up to you to write the code for computing this color from all the attributes that you attached to the vertices

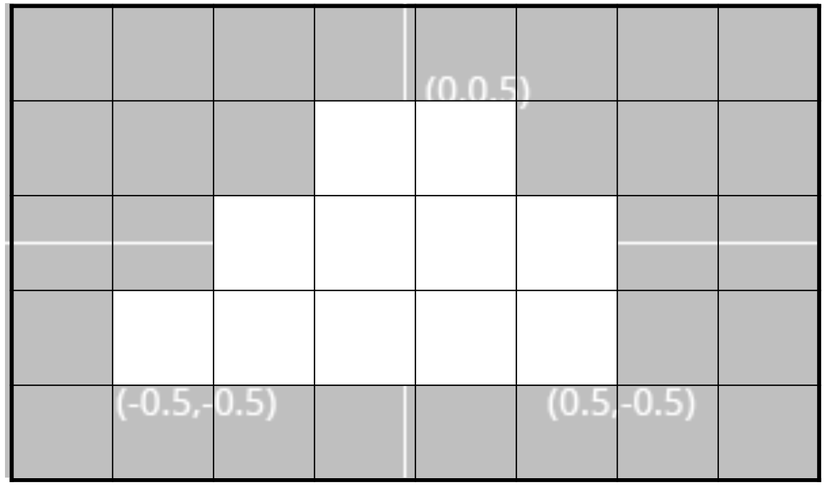
Vertex Shader

Fragment Shader



A simple fragment shader

```
vector<VertexAttributes> vertices;
vertices.push_back(VertexAttributes(-0.5,-0,5));
vertices.push_back(VertexAttributes(0.5,-0.5,0));
vertices.push_back(VertexAttributes(0,0.5,0));
```



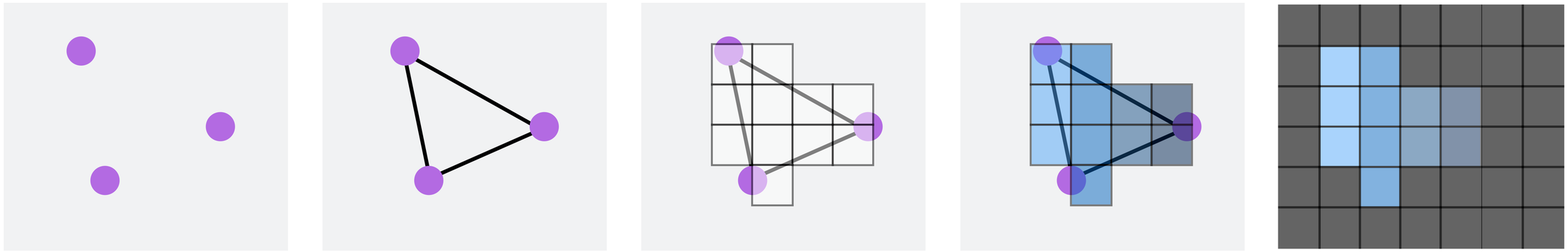

```
program.FragmentShader = [](const VertexAttributes& va, const UniformAttributes& uniform)
{
    return FragmentAttributes(1,1,1);
};
```

The colors produced by the fragment shader must be between 0.0 and 1.0

Rasterization Pipeline

Input
Vertex Attributes
Uniform Attributes

Output
RGBA Image



Vertex Shader

Primitive Assembly

Rasterization

Fragment Shader

Blending Shader

Blending Shader

- The blending shader decides how to blend a fragment with the colors/attributes already present on the corresponding pixel of the framebuffer

```
// The blending shader converts colors between 0 and 1 to uint8
program.BlendingShader = [](const FragmentAttributes& fa, const FrameBufferAttributes& previous)
{
    return FrameBufferAttributes(fa.color[0]    ,fa.color[1]    ,fa.color[2]    ,fa.color[3]    );
};
```

We are all set!

- After specifying the 3 shaders (vertex, fragment, blending) we can rasterize the vertices (every 3 vertices form a triangle):

```
rasterize_triangles(program,uniform,vertices,frameBuffer);
```

- Finally, the framebuffer can be saved as a png as we did for raytracing

```
vector<uint8_t> image;
framebuffer_to_uint8(frameBuffer,image);
stbi_write_png("triangle.png", frameBuffer.rows(), frameBuffer.cols(), 4, image.data(), frameBuffer.rows()*4);
```

Attributes

- The rasterizer uses an additional file called attributes.h to define the struct used for vertex, fragment, and framebuffer attributes
- You will have to change this depending on what informations you want your shaders to have access to

Vertex Attributes

```
class VertexAttributes
{
    public:
    VertexAttributes(float x = 0, float y = 0, float z = 0, float w = 1)
    {
        position << x,y,z,w;
    }

    // Interpolates the vertex attributes
    static VertexAttributes interpolate(
        const VertexAttributes& a,
        const VertexAttributes& b,
        const VertexAttributes& c,
        const float alpha,
        const float beta,
        const float gamma
    )
    {
        VertexAttributes r;
        r.position = alpha*a.position + beta*b.position + gamma*c.position;
        return r;
    }

    Eigen::Vector4f position;
};
```

Fragment Attributes

```
class FragmentAttributes
{
    public:
    FragmentAttributes(float r = 0, float g = 0, float b = 0, float a = 1)
    {
        color << r,g,b,a;
    }

    Eigen::Vector4f color;
};
```

FrameBuffer Attributes

```
class FrameBufferAttributes
{
    public:
    FrameBufferAttributes(uint8_t r = 0, uint8_t g = 0, uint8_t b = 0, uint8_t a = 255)
    {
        color << r,g,b,a;
    }

    Eigen::Matrix<uint8_t,4,1> color;
};
```

Uniforms

- Uniform are values that are constant for the entire **scene**, i.e. they are not attached to vertices
- They are essentially global variables within the shaders
- All vertices and all fragments will see the same value
- For example, let's change the demo code to use a uniform to store the triangle color

Some Tips

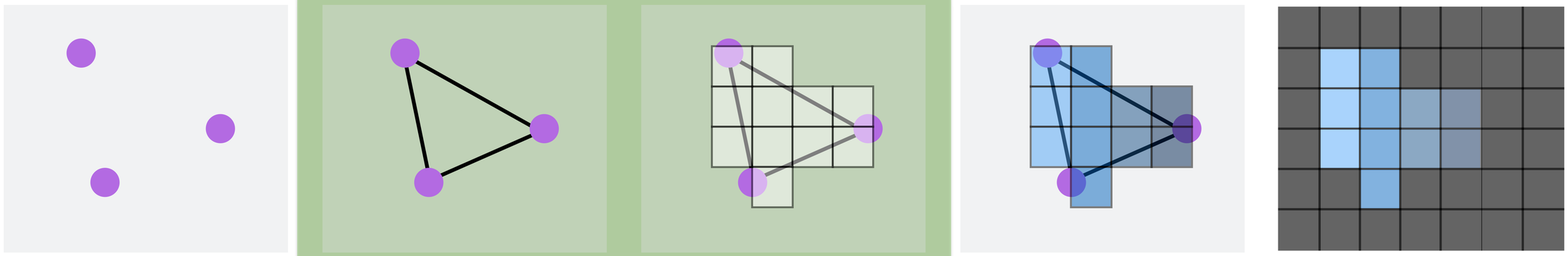
- Start from a **WORKING** application and do your changes incrementally.
- You can add breakpoints inside raster.h/.cpp to check what is going wrong
- You can also export the framebuffer to a png at any point in the code to see what is going on

Software Rasterization Examples

Rasterization Pipeline

Input
Vertex Attributes
Uniform Attributes

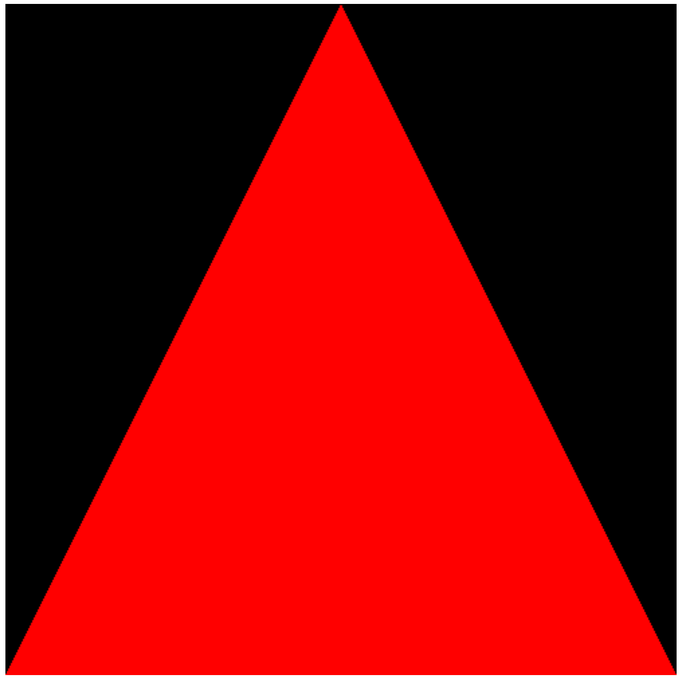
Output
RGBA Image



Live Coding Session

- We will code together a few demos to showcase different features and concepts that we studied in the last lecture
- No need to take notes, all the source files are available in the “extra” folder of Assignment 5

Starting Code



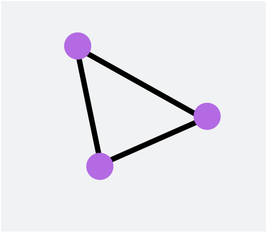
Supported Primitives

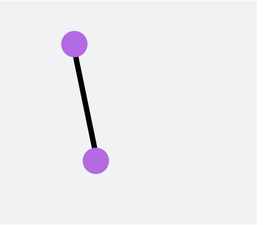
```
// Rasterizes a single triangle v1,v2,v3 using the provided program and uniforms.
// Note: v1, v2, and v3 needs to be in the canonical view volume (i.e. after being processed by the vertex shader)
void rasterize_triangle(const Program& program, const UniformAttributes& uniform, const VertexAttributes& v1, const
VertexAttributes& v2, const VertexAttributes& v3, FrameBuffer& frameBuffer);

// Rasterizes a collection of triangles, assembling one triangle for each 3 consecutive vertices.
// Note: the vertices will be processed by the vertex shader
void rasterize_triangles(const Program& program, const UniformAttributes& uniform, const std::vector<VertexAttributes>&
vertices, FrameBuffer& frameBuffer);

// Rasterizes a single line v1,v2 of thickness line_thickness using the provided program and uniforms.
// Note: v1, v2 needs to be in the canonical view volume (i.e. after being processed by the vertex shader)
void rasterize_line(const Program& program, const UniformAttributes& uniform, const VertexAttributes& v1, const
VertexAttributes& v2, float line_thickness, FrameBuffer& frameBuffer);

// Rasterizes a collection of lines, assembling one line for each 2 consecutive vertices.
// Note: the vertices will be processed by the vertex shader
void rasterize_lines(const Program& program, const UniformAttributes& uniform, const std::vector<VertexAttributes>&
vertices, float line_thickness, FrameBuffer& frameBuffer);
```

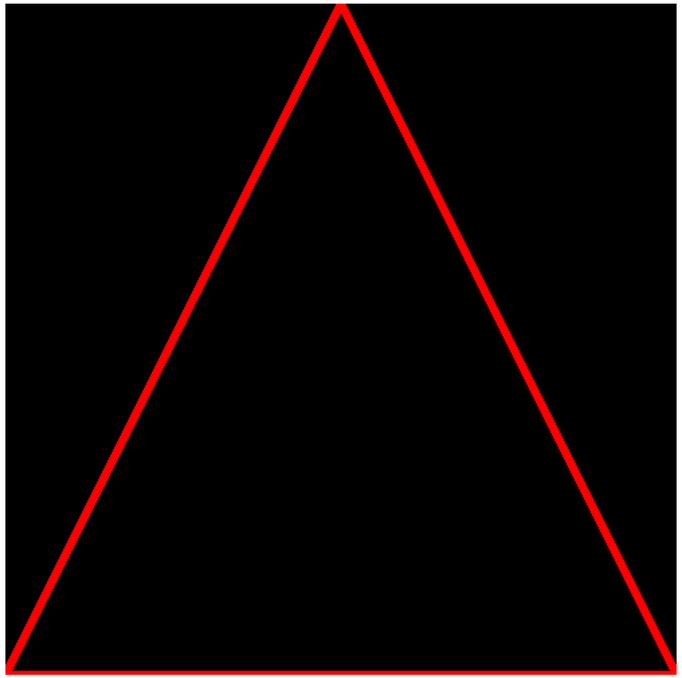




Let’s take a detailed look at how these 4 functions work

Line Rasterization

- Let us draw a red border instead of a triangle
- **python rename.py lines**



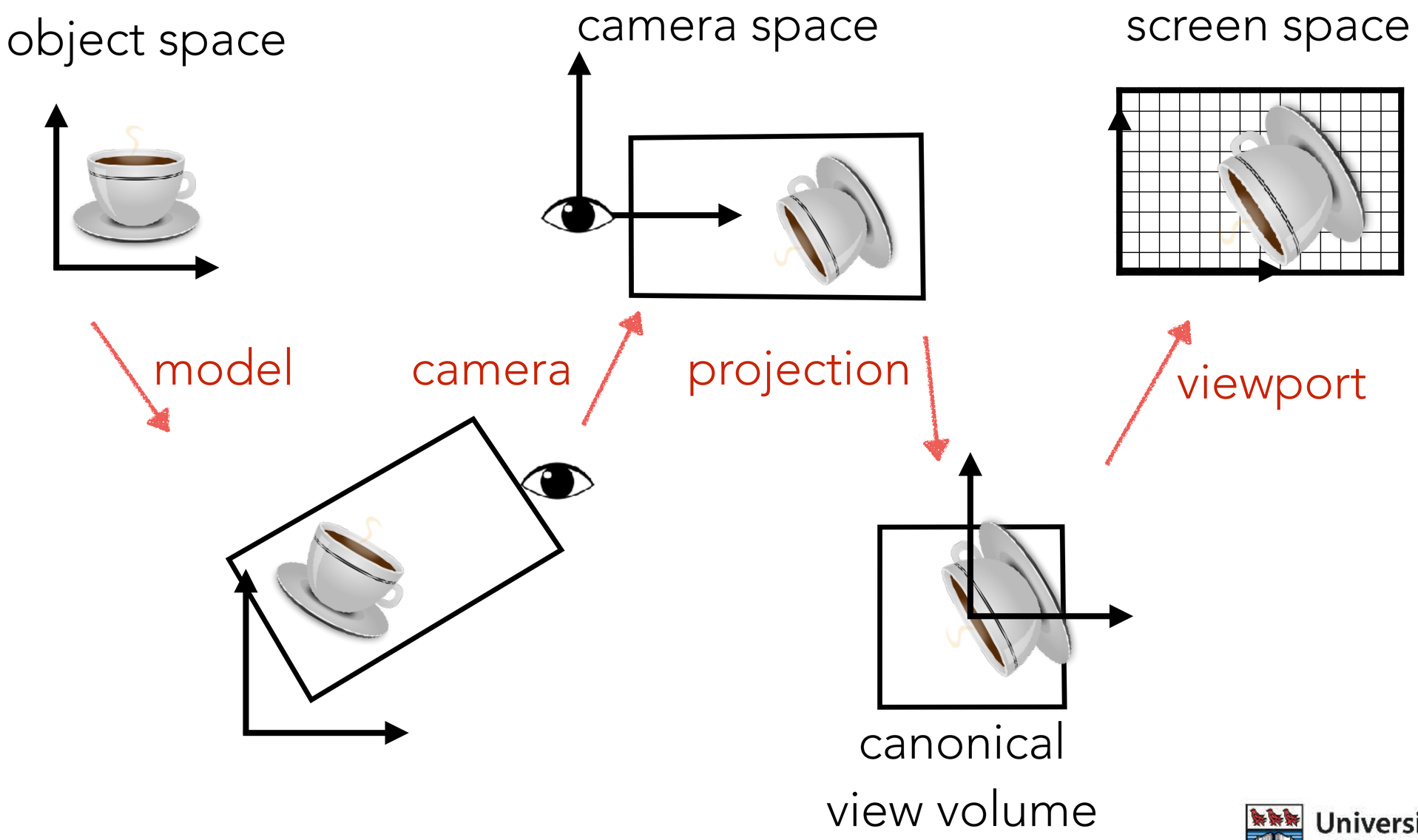
Vertex Attributes

- Let us add a color vertex attribute and interpolate it inside a triangle
- **python rename.py attributes**

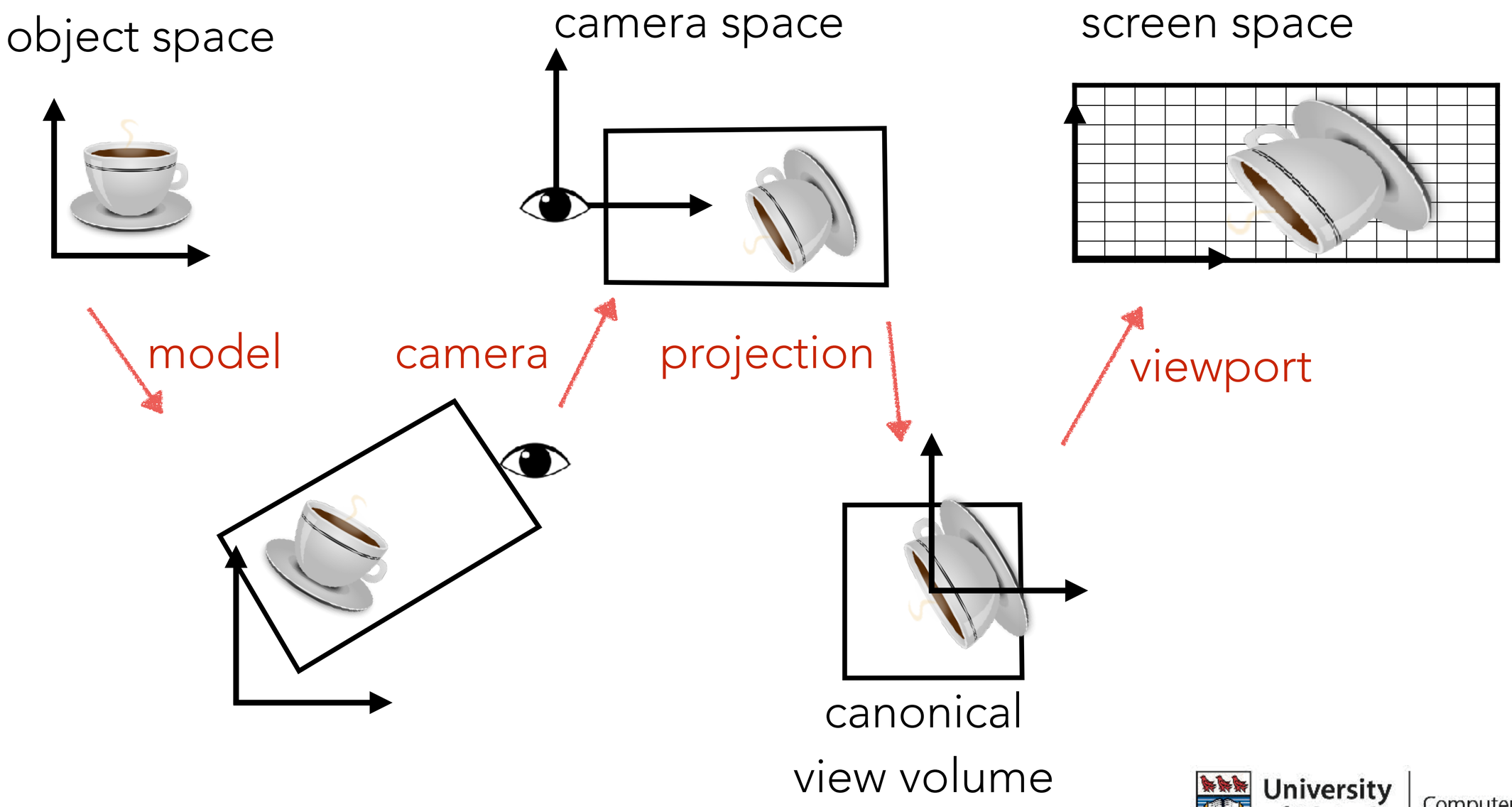
View/Model Transformation

- View/Model transformations are applied in the vertex shader
- They are passed to the shader as uniform (global variables)
- To create transformation matrices you can do it by hand or use the geometry module of Eigen: https://eigen.tuxfamily.org/dox/group__TutorialGeometry.html

Viewport and View Transformation



Viewport and View Transformation

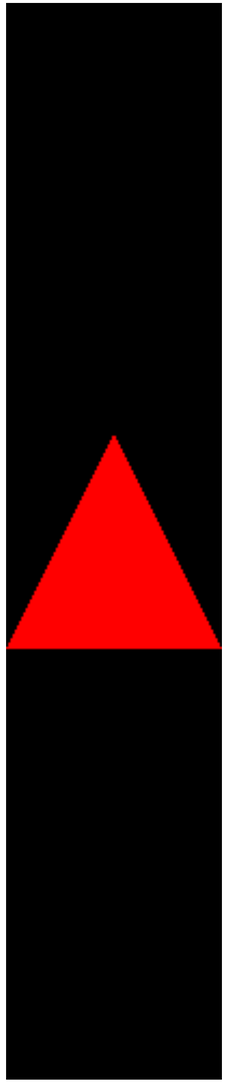


How to prevent viewport distortion?

- We need to adapt the view depending on the size of the framebuffer
- We can then create a view transformation that maps a box with the same aspect-ratio of the viewport into the unit cube
- Equivalently, we are using a “camera” that has the same aspect ratio as the window that we use for rendering
- In this way, the distortion introduced by the viewport transformation will cancel out

View Transformation

- Let's add a view transformation to prevent viewport distortion
- **python rename.py view**

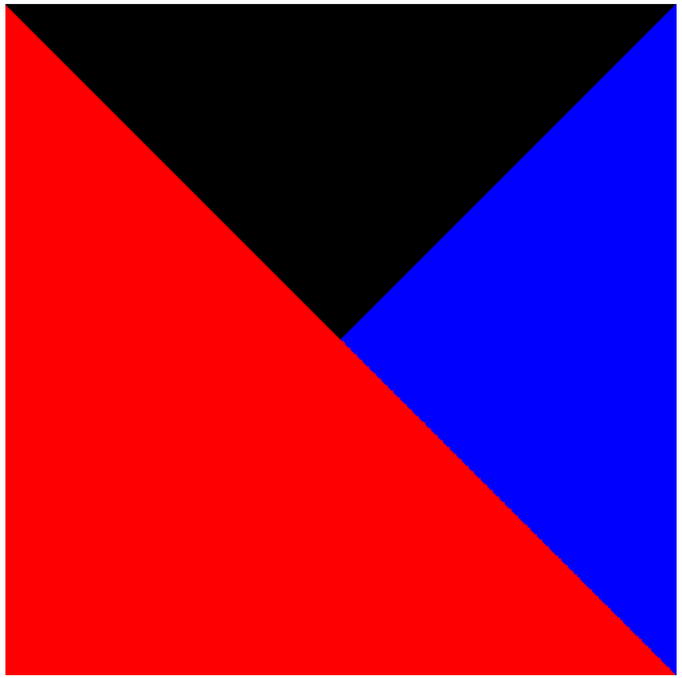


Tests (Special case of Blending)

- The tests determine if a fragment will affect the color in the framebuffer or if it should be discarded
- There are main use cases:
  - Depth Test - Discards all fragments with a z coordinate *bigger* than the value in the depth buffer
  - Stencil Test - Discards all fragments outside a user-specified mask

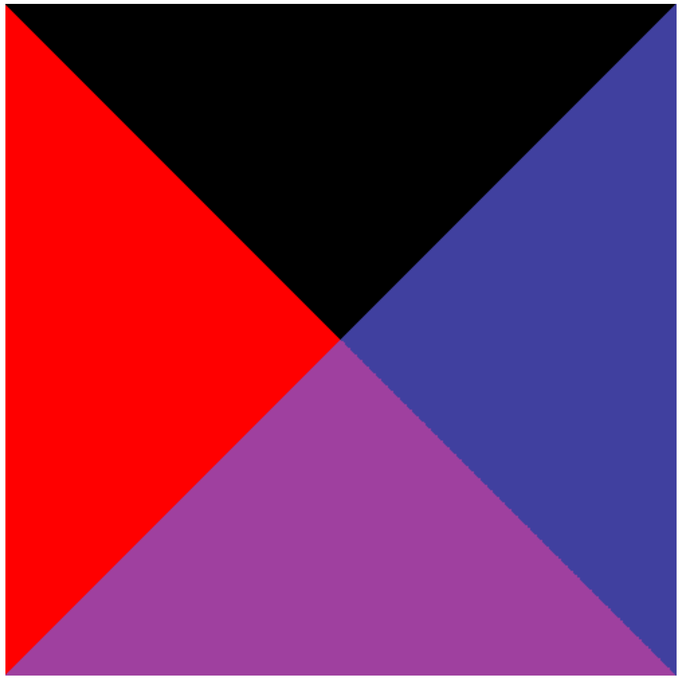
Depth Test

- Let's draw a couple of triangles with depth test active
- **python rename.py depth**



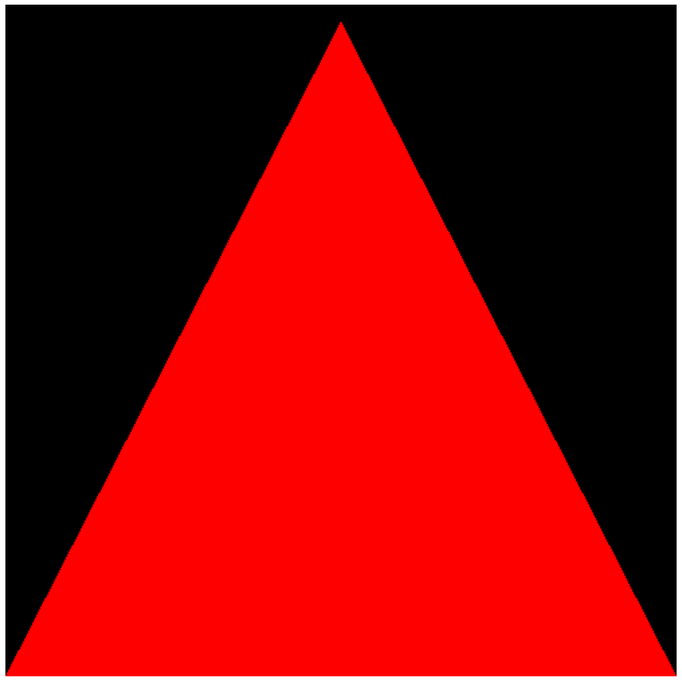
Blending

- Let's see how to render a transparent triangle
- **python rename.py depth**



Animation

- Let's see how to render a simple animation
- **python rename.py animation**



Supersampling

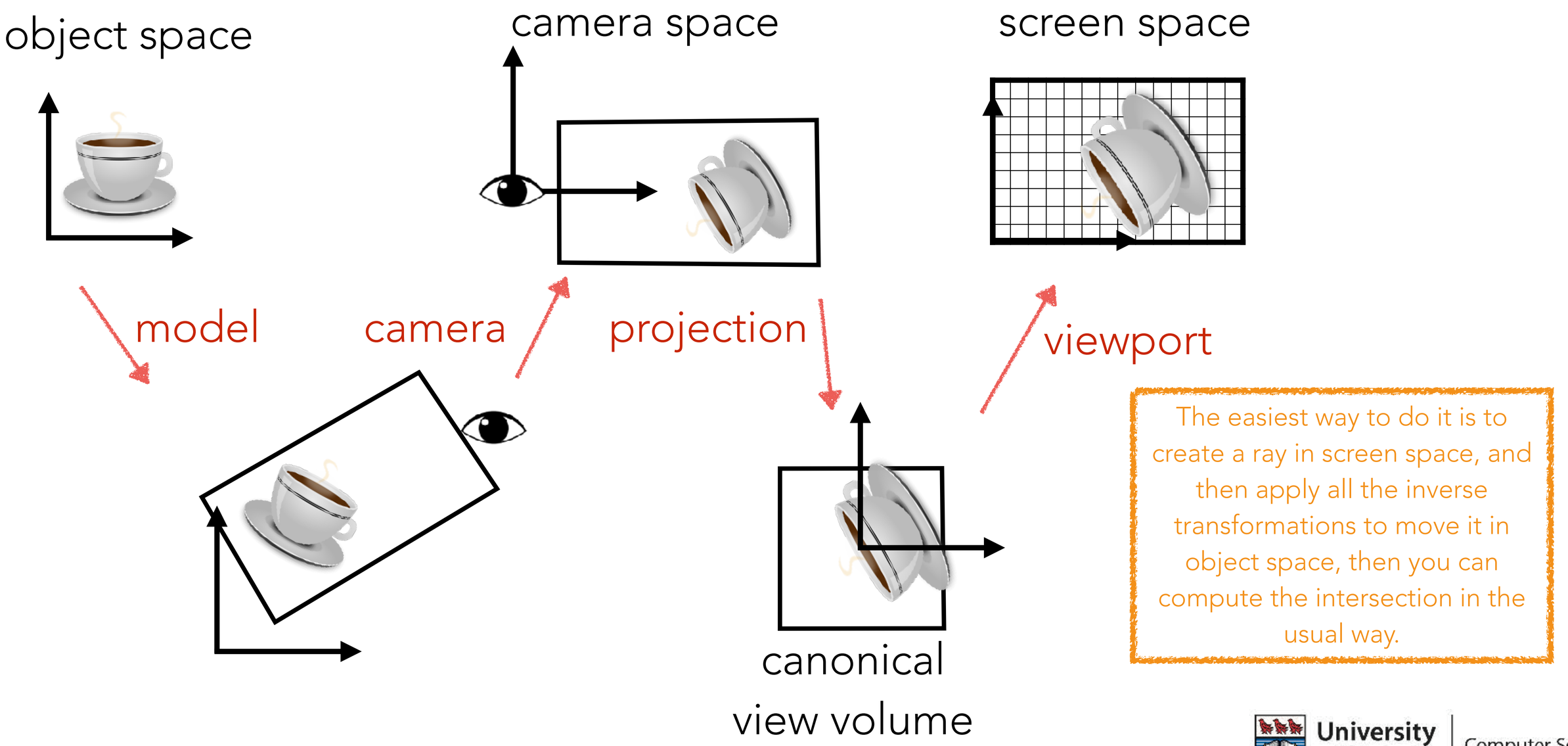
- We can render an image 4 times larger and then scale it down to avoid sharp edges

- **python rename.py supersampling**

Picking Objects

- To interact with the scene (for the final project), it is common to “pick” or “select” objects in the scene
- The most common way to do it is to cast a ray, starting from the point where the mouse is and going “inside” the screen
- The first object that is hit by the ray is going to be the selected object
- For picking in the exercises, you can reuse the code that you developed in the first assignment
- You must account for viewing and model transformations!

Picking via Ray Casting



The easiest way to do it is to create a ray in screen space, and then apply all the inverse transformations to move it in object space, then you can compute the intersection in the usual way.

References

Fundamentals of Computer Graphics, Fourth Edition
4th Edition **by Steve Marschner, Peter Shirley**

Chapter 17