UV coordinates and Pipeline creation

Texture

Appearance of 3D objects should be defined in great detail to make them realistic, since, in most of the cases, material specification varies from point to point.

Assigning a different material to a large number of very small triangles is not practical in most of the cases due to memory requirements.

The common approach uses tables to assign different values to the parameters of the shaders, depending on the positions of the internal points of the surface.

In the most common case, such tables contain the diffuse color of the object, which is acquired from an image.

The considered tables (or images) are called **textures** or *maps*.

Texture

Example of use of texture effects can be seen in 3D games for the consoles of the mid-90s' (i.e. *Sony Playstation I*).

Tomb Raider (Eidos Interactive - 1996)

Texture

Textures can be either:

- 1D Texture
- 2D Texture
- 3D Texture
- Cube map

Texture

2D textures define the parameters of the surface of an object.

3D textures also define parameters for a volume.

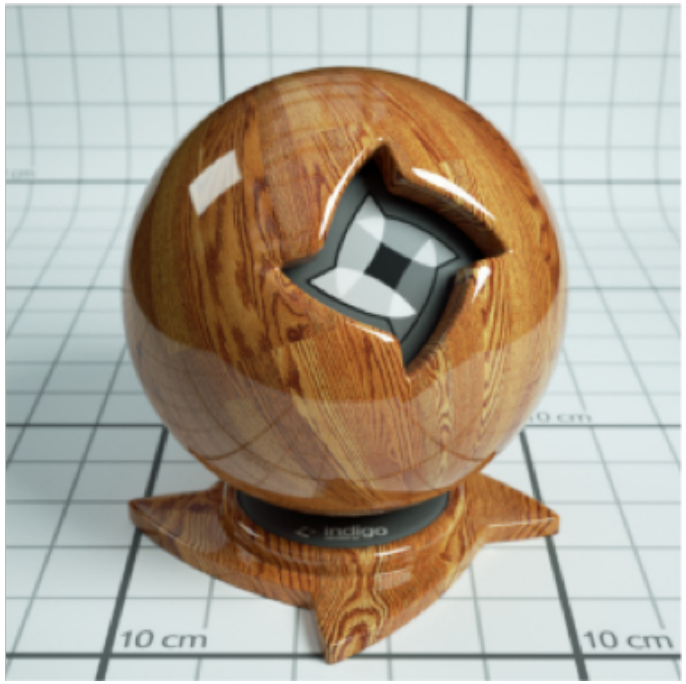
1D textures are instead used to contain values of pre-computed functions, or to perform optimizations.

Cube map textures are instead used to define parameters associated with *directions* in a 3D coordinate system. They use a slightly different approach, and will be introduced later.

Texture

Images that define the surface of an object are planar.

However the objects on which they are applied, are characterized by complex non-planar topologies, composed of several sides.

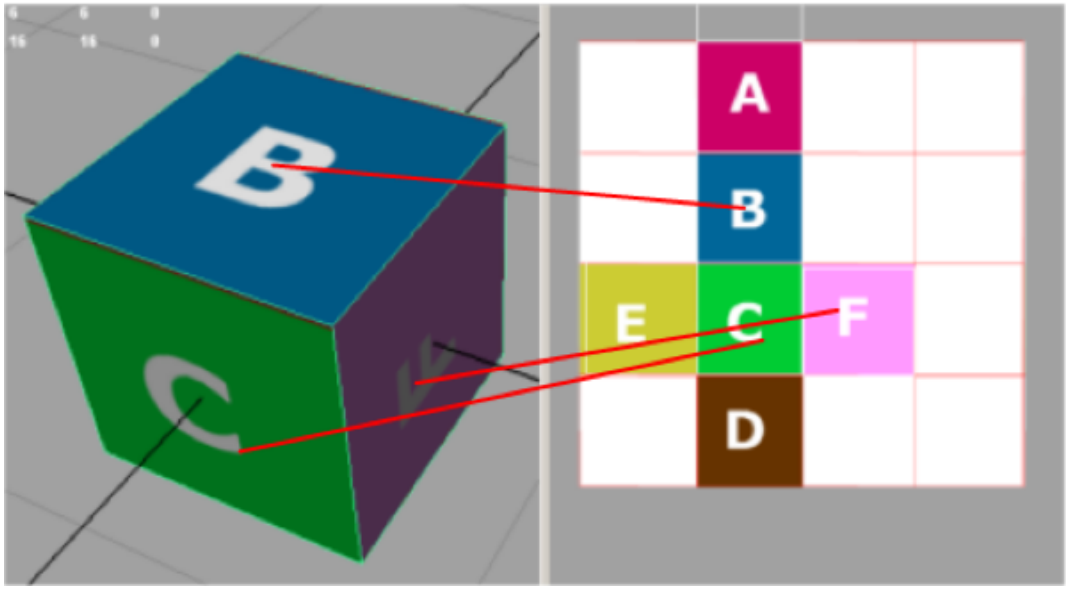


Texture

2D textures are applied to 3D objects, using a *mapping relation* that associates each point on the surface with a point on the texture.

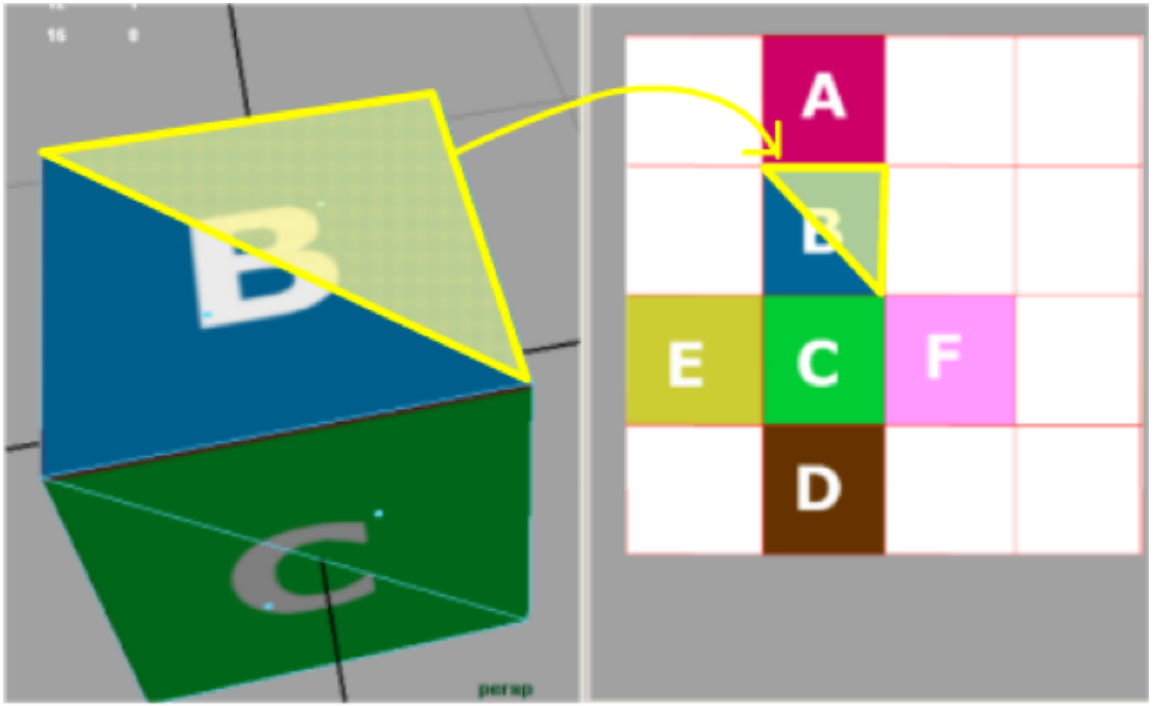
The mapping creates a correspondence between pixels of the texture (called *texel*) and points of the object.

Note that in general it is very difficult to create a mapping that uses the entire space available on the texture, and some unused areas are usually present.



Texture

In case of polygonal objects, the mapping procedure creates a correspondence between triangles on the mesh, and triangles over the texture.

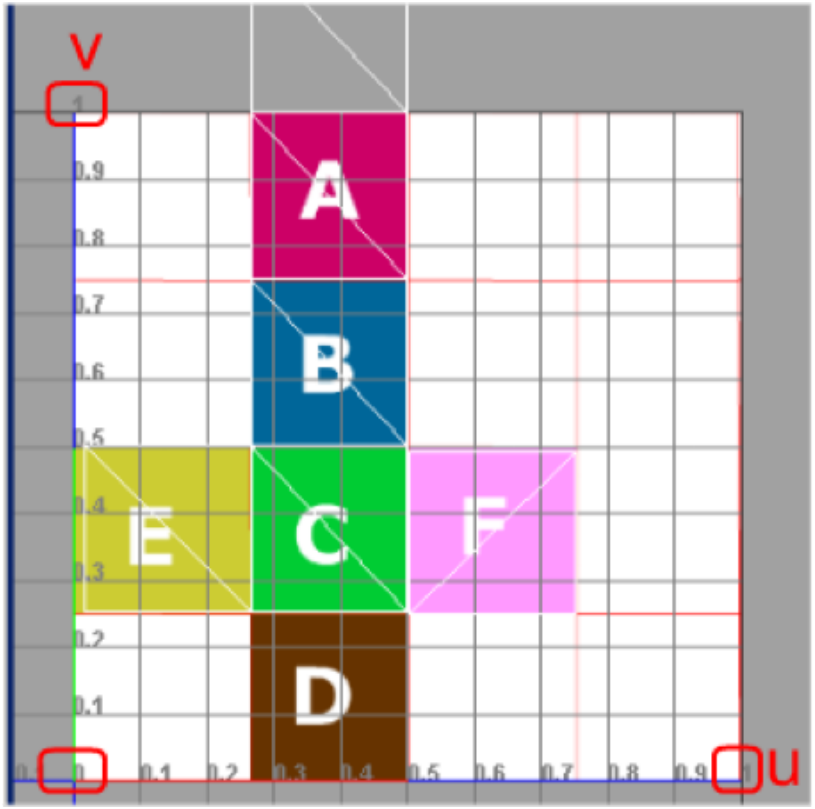


UV coordinates

Points over 2D Textures are addressed using a Cartesian coordinate system, whose axes are called *u* and *v*.

This set of coordinates are usually called *UV, mapping or texture coordinates*.

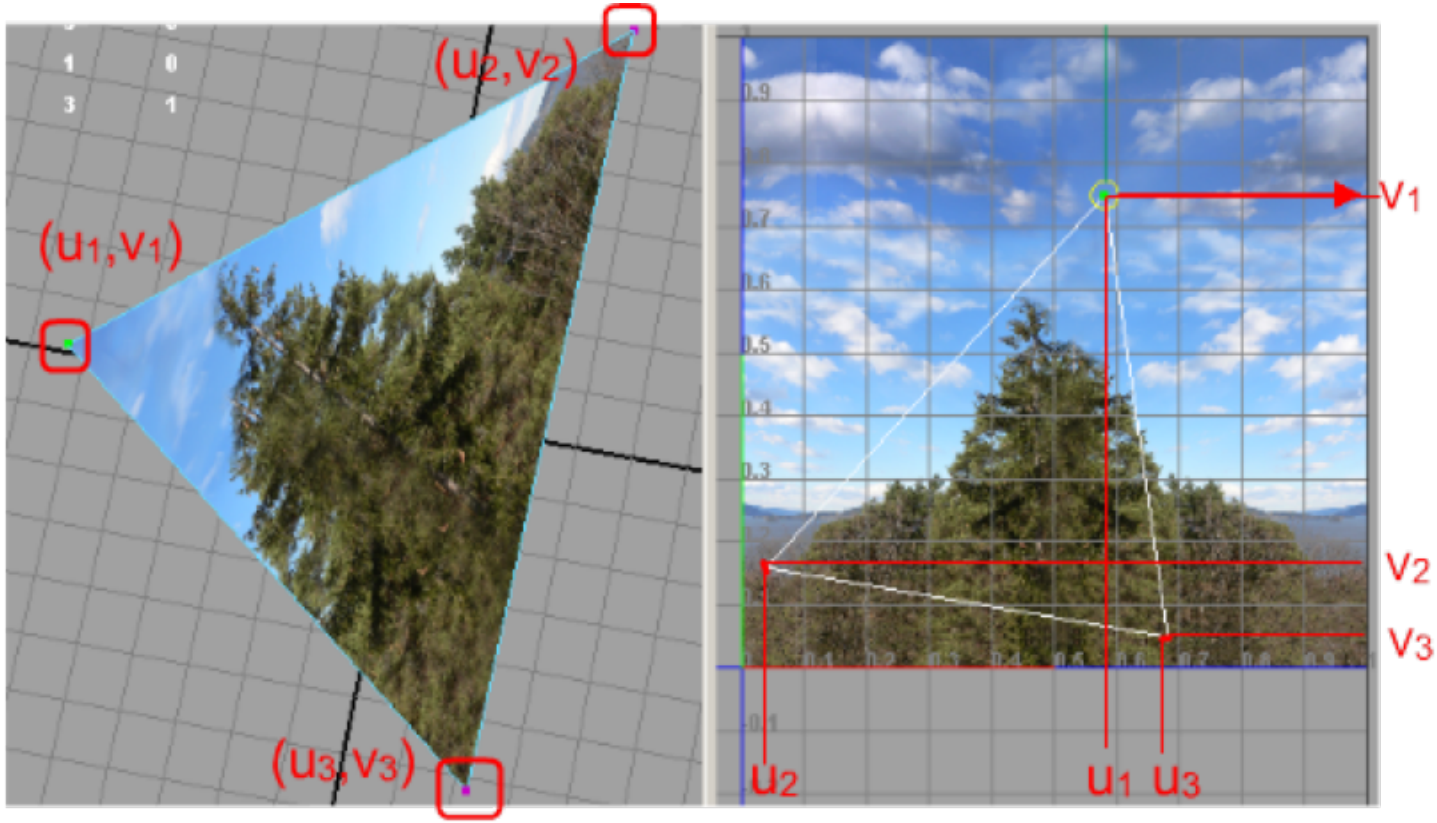
As a convention, the *u* and *v* values vary between *0* and *1* along the horizontal and vertical axes of the texture.



Note: sometimes an alternative axis naming convention, namely ST, is used. In some context, both notations are used with different meanings. In this course we will not enter in this detail, and simply use the UV notation.

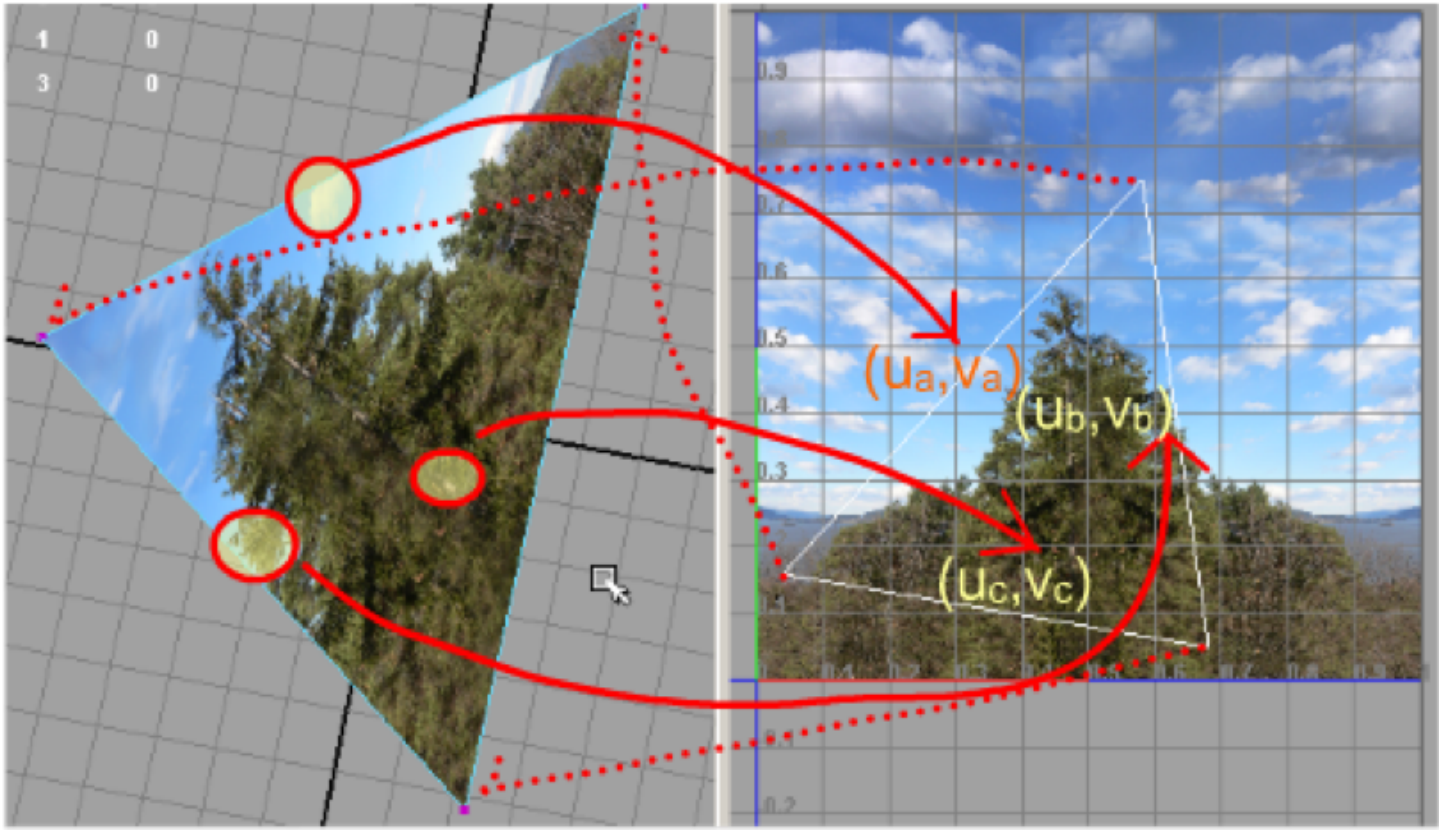
UV coordinates

UV coordinates are assigned only to the vertices of the triangles.



UV coordinates

For the internal points, the values of the UV coordinates are computed via interpolation.

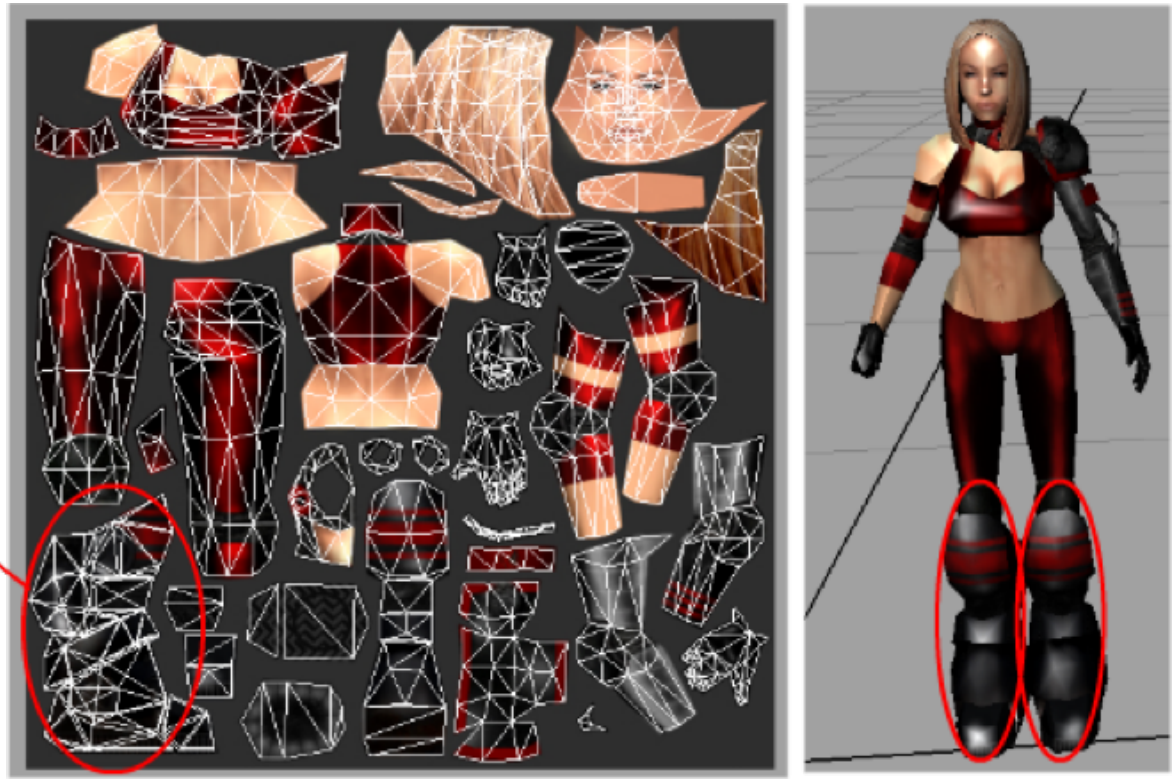


UV coordinates

The proper assignment of UV coordinates to the model is a very important but complex (long and tedious) operation performed by 3D artists and modelers.

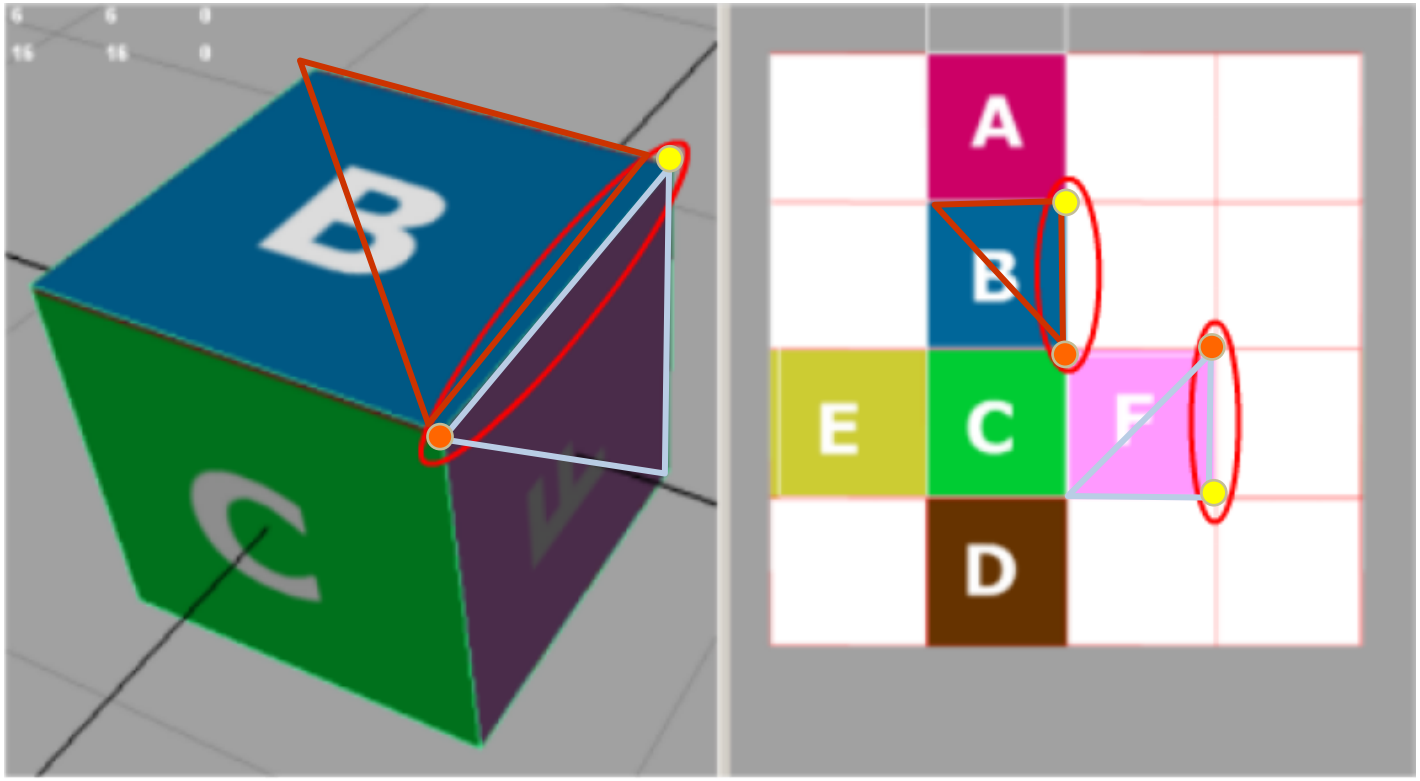
UV coordinates

UV mapping does not need to be a bijection: the same part of the texture can be shared by several triangles on the 3D object. Elite modelers can exploit this feature to improve the quality of their models.



UV coordinates

Note that, as seen for normal vector directions, two vertices belonging to two different triangles may have the same position in space, but different UV coordinates.



UV coordinates

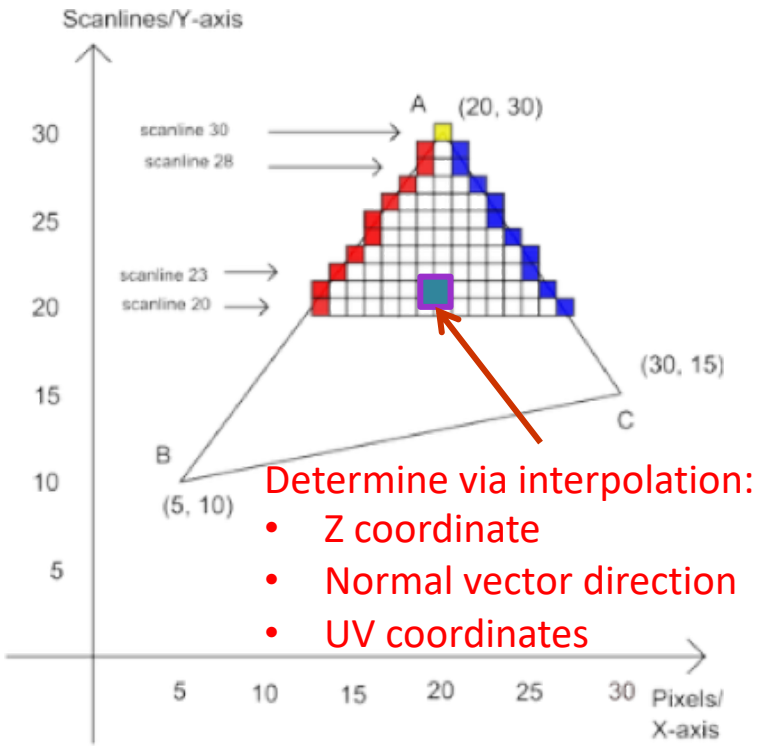
For example, in a cylinder where a rectangle is “wrapped” around its side, the vertices at the beginning and the end must be replicated: in this case, we will have two vertices with the same position, same normal vector direction, but different UV coordinates.

Rasterisation with textures

The first step to support textures is the interpolation of the UV coordinates together with the other parameters of the vertices (i.e. the normal vectors).

Then a lookup procedure is used to gather the corresponding texel.

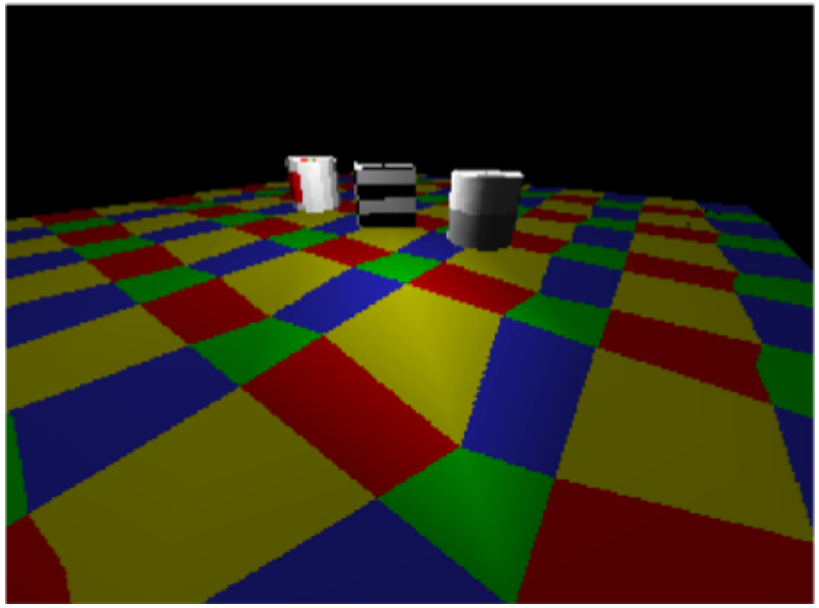
Finally a per-pixel operation is used to compute the rendering equation for the pixel on screen.



Perspective interpolation

If perspective is used, interpolation cannot be done in the conventional way due to the non-linearity of the projection.

The image below shows what happens applying conventional interpolation to compute the internal values of parameters associated to triangles when using perspective: straight lines appear to be wobbling.



Perspective interpolation

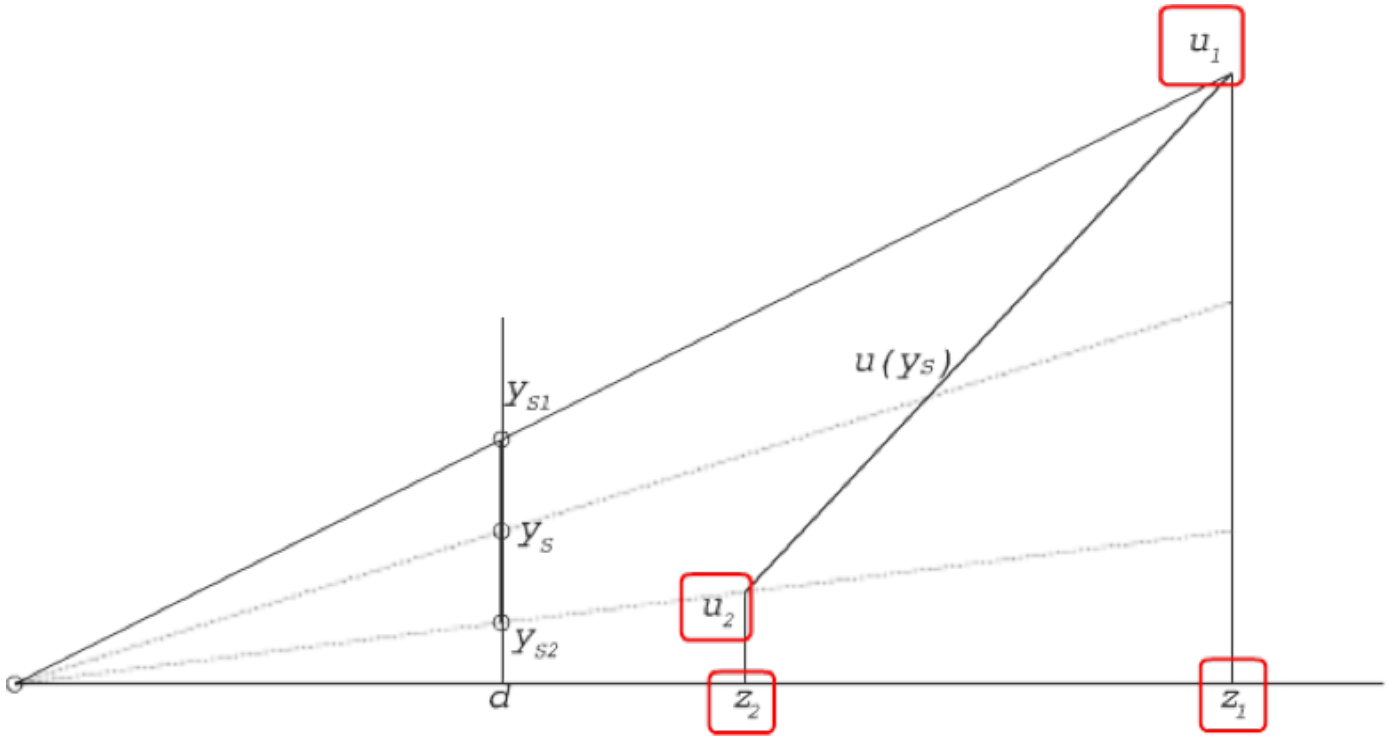
To avoid such wobbling effects, applications use *perspective correct interpolation*.

With this technique, interpolation is non-linear, and depends on the distance of the interpolated points from the projection plane.

Although outside the scope of this course, the perspective correct interpolation will briefly described for its theoretical importance, and for its impact on the overall performance of an application.

Perspective correction: derivation

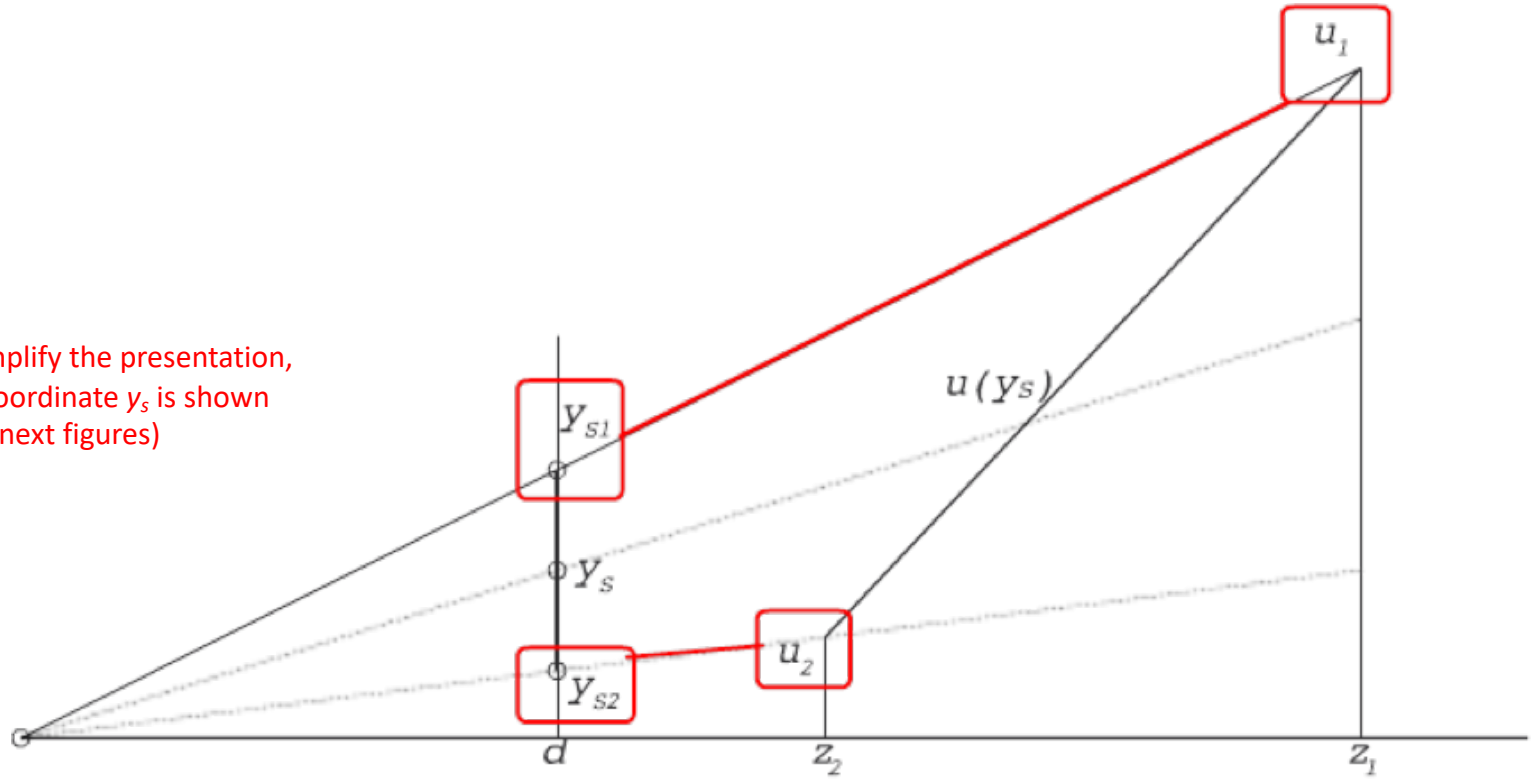
Let us consider a parameter u, which can be any value associated either to a texture coordinate, a color or a normal vector direction, and which assumes values u_1 at distance z_1, and u_2 at z_2.



Perspective correction: derivation

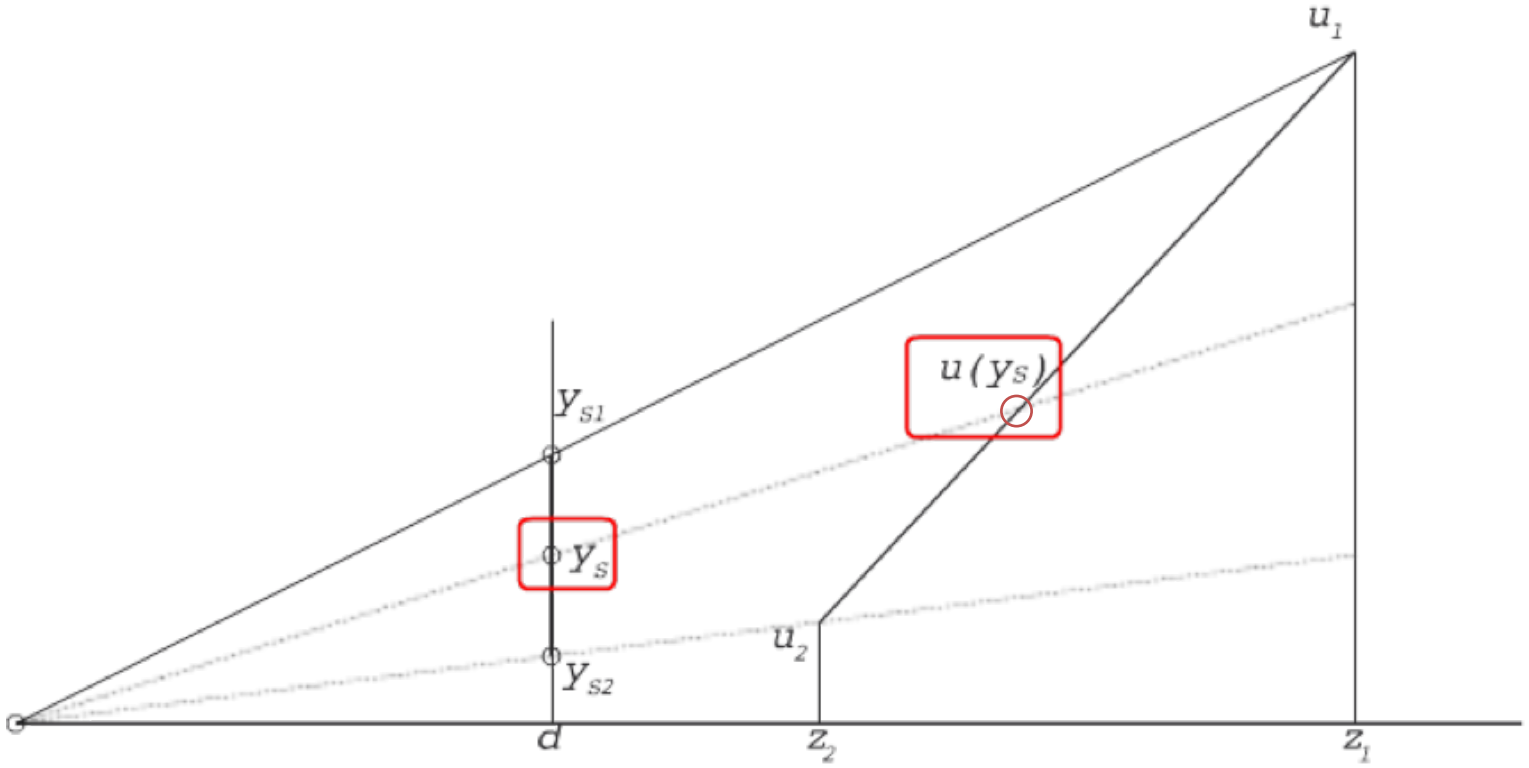
Let us consider a parameter u: at point (x_1, y_1, z_1) it assumes value u_1 , and at (x_2, y_2, z_2) value u_2 . The two points are projected to screen coordinates (x_{S1}, y_{S1}) and (x_{S1}, y_{S2}).

(to simplify the presentation, only coordinate y_s is shown in the next figures)



Perspective correction: derivation

We are interested in determining the perspective correct value $u(x_S, y_S)$ of the parameter u in 3D space, corresponding to a pixel at coordinate (x_S, y_S) on screen.



Perspective correction: derivation

Let us call α the interpolation value that computes y_S from y_{S1} and y_{S2}

$$y_S = y_{S1}\alpha + y_{S2}(1-\alpha)$$

$$\alpha = \frac{y_S - y_{S2}}{y_{S1} - y_{S2}}$$

Perspective correction: derivation

Perspective correct interpolation uses the formula below.

In practice, it interpolates the value of the parameters *u* divided by the distance *z*.

And then brings back everything to its correct dimension dividing by the interpolated version of 1/z .

$$u(\alpha) = \frac{\alpha \frac{u_1}{z_1} + (1-\alpha)\frac{u_2}{z_2}}{\frac{\alpha}{z_1} + \frac{(1-\alpha)}{z_2}}$$

Perspective correction: derivation

As introduced, an internal point (x_S,y_S) of a triangle can be considered as a linear combination its three vertices (x_1,y_1), (x_2,y_2), (x_3,y_3) with coefficients that sums up to one.

$$(x_S, y_S) = (x_1, y_1) \cdot \alpha_1 + (x_2, y_2) \cdot \alpha_2 + (x_3, y_3) \cdot \alpha_3$$

with:

$$\alpha_1 + \alpha_2 + \alpha_3 = 1$$

If we call u_1, u_2 and u_3, the value of a parameter u at the three vertices, the value u_s at (x_S,y_S) with perspective correct interpolation can thus be computed as:

$$u_S = u(\alpha_1, \alpha_2, \alpha_3) = \frac{\alpha_1 \frac{u_1}{z_1} + \alpha_2 \frac{u_2}{z_2} + \alpha_3 \frac{u_3}{z_3}}{\frac{\alpha_1}{z_1} + \frac{\alpha_2}{z_2} + \frac{\alpha_3}{z_3}}$$

Perspective correction: derivation

Perspective correct interpolation should be performed for all parameters considered *after the perspective projection*.

As seen, perspective correct interpolation requires the distance of the point from the center of projection along the negative z axis: in clipping coordinates (i.e. before the normalization step), this corresponds to the fourth component of the homogenous coordinate vector changed of sign.

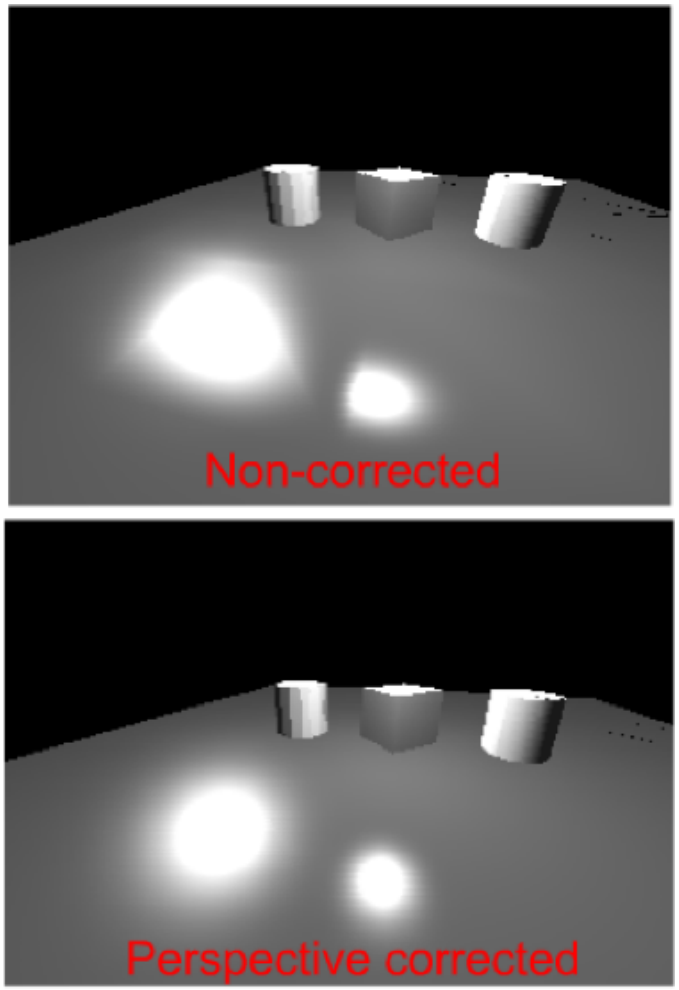
$$P_{persp} \cdot v = \begin{vmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{vmatrix} \cdot \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix} = \begin{vmatrix} x_C \\ y_C \\ z_C \\ \boxed{-z} \end{vmatrix}$$

Perspective interpolation

Note that the problem of non linear interpolation of values does not apply only to UV coordinates, but also to colors and normal vectors.

However, for those parameters it is usually less perceivable, and linear approximation can sometimes be used as a faster alternative.

Moreover, as for the Goureaud shading technique, the problem is less visible with very detailed meshes.



Interpolation between Vertex and Fragment shader

The default interpolation between Vertex and Fragment shader is via Perspective Correct interpolations.

However it can be controlled with the `flat` and `noperspective` directives before the `in` and `out` variables.

Vertex shader

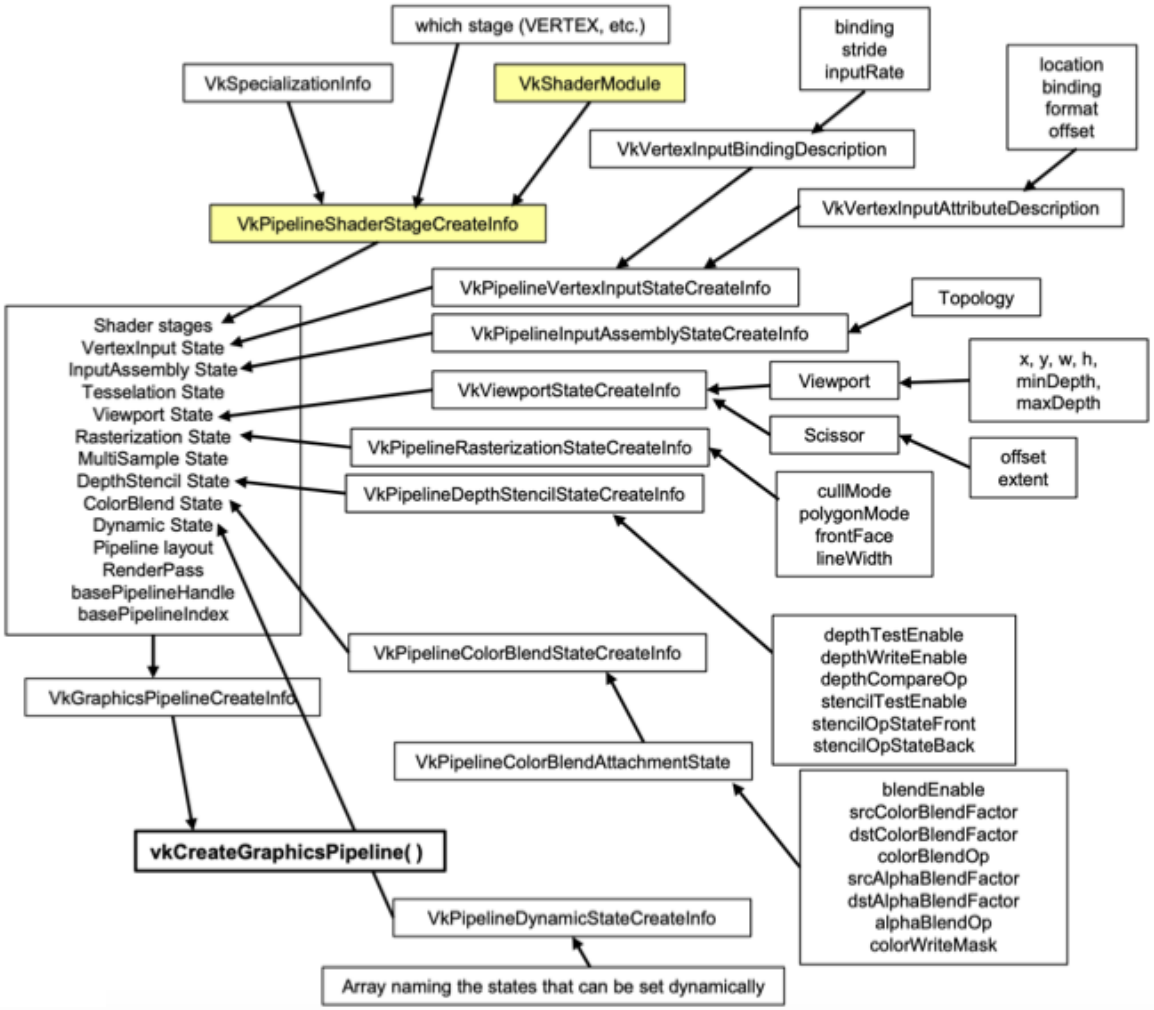
```
layout(location = 0) out vec3 fragPos;
layout(location = 1) out vec3 fragNorm;
layout(location = 2) noperspective out vec2 fragUV;
```

Fragment shader

```
layout(location = 0) in vec3 fragPos;
layout(location = 1) in vec3 fragNorm;
layout(location = 2) noperspective in vec2 fragUV;
```

Pipeline creation

The Vulkan pipeline is the most complex and important data structure of the entire 3D visualization process.

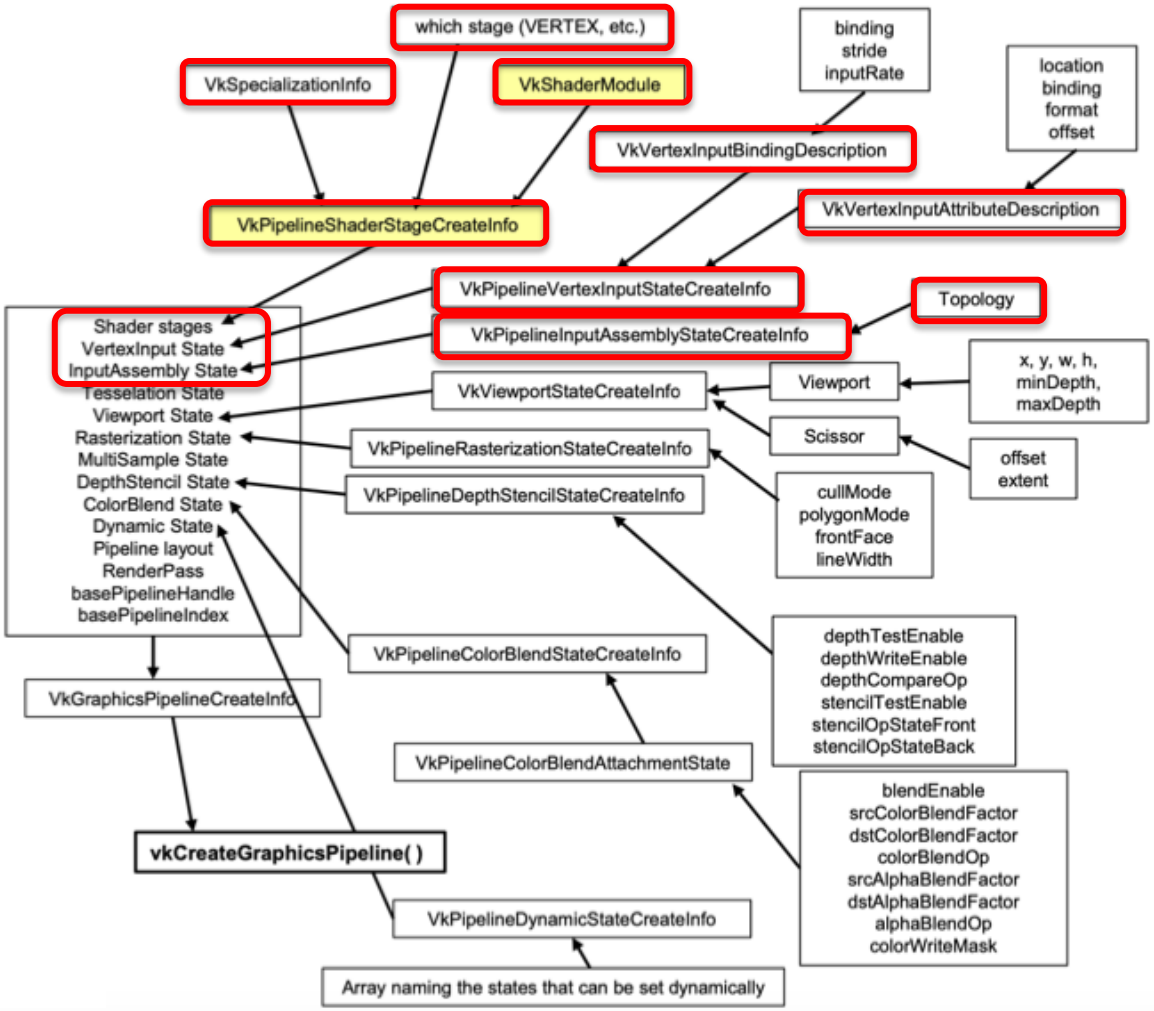


From: https://web.engr.oregonstate.edu/~mjb/vulkan/

Pipeline creation

We have already seen how to configure the *Input Assembler*, the *Vertex Input*, and the *Shader stages*.

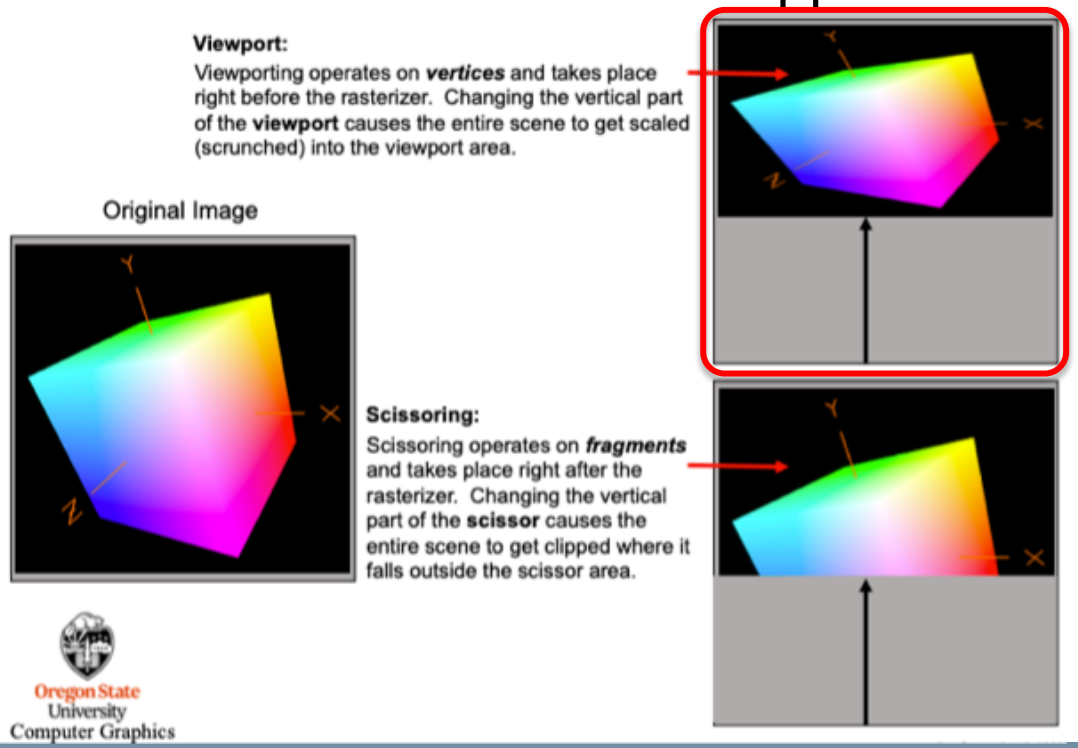
Now we will see how to complete the pipeline creation process.



From: https://web.engr.oregonstate.edu/~mjb/vulkan/

Viewport

Viewport defines a rectangular area on the surface that will be used by the rendering system: i.e. where the (-1,-1) and (1,1) normalized screen coordinates will be mapped.



Viewport

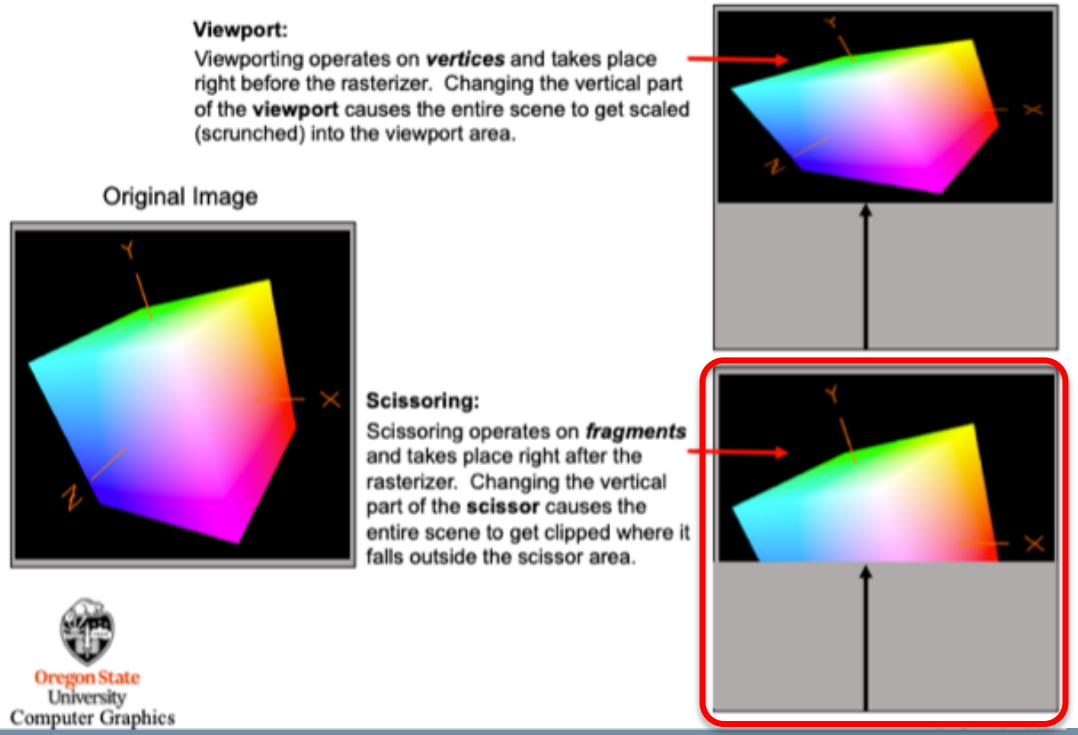
It is defined with a `VkViewport` object, where the (`x`, `y`) coordinates of the top-left corner and the `width` and height `of` the viewport, all specified in pixel are assigned to the corresponding fields. If the whole are needs to be used, we can read these value from the surface we created.

The range of the normalized *z* screen coordinates can also be specified in the `minDepth` and `maxDepth` field.

```
VkViewport viewport{};
viewport.x = 0.0f;
viewport.y = 0.0f;
viewport.width = (float) extent.width;
viewport.height = (float) extent.height;
viewport.minDepth = 0.0f;
viewport.maxDepth = 1.0f;
```

Scissor test

If we need to leave some are of the screen unaffected by Vulkan (for example to use it to draw some GUI inside), we can “cut” rectangular area using *Scissor rectangles*.



Scissor test

In this case we specify the position in pixel to cut in a `VkRect2D` structure.

If we want to use the full window, we can get these value from the surface as done before with the viewport.

```
VkRect2D scissor{};
scissor.offset = {0, 0};
scissor.extent = extent;
```

Viewport State structure

Both viewport and scissor are combined into a VkPipelineViewportStateCreateInfo structure that will be used when creating the pipeline.

```
VkPipelineViewportStateCreateInfo viewportState{};
viewportState.sType = VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;

viewportState.viewportCount = 1;
viewportState.pViewports = &viewport;
viewportState.scissorCount = 1;
viewportState.pScissors = &scissor;
```

Back-face culling

The rasterizer component of the pipeline, defined inside a `VkPipelineRasterizationStateCreateInfo` object, controls both back-face culling, and other parameters such as the possibility of drawing wireframe primitives, and the line width for segments (not supported on all hardware).

```
VkPipelineRasterizationStateCreateInfo rasterizer{};
rasterizer.sType =
          VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
rasterizer.depthClampEnable = VK_FALSE;
rasterizer.rasterizerDiscardEnable = VK_FALSE;
rasterizer.polygonMode = VK_POLYGON_MODE_FILL;
rasterizer.lineWidth = 1.0f;
rasterizer.cullMode = VK_CULL_MODE_BACK_BIT;
rasterizer.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
rasterizer.depthBiasEnable = VK_FALSE;
rasterizer.depthBiasConstantFactor = 0.0f; // Optional
rasterizer.depthBiasClamp = 0.0f; // Optional
rasterizer.depthBiasSlopeFactor = 0.0f; // Optional
```

Back-face culling

Field `polygonMode` controls if we want to draw filled triangles (`VK_POLYGON_MODE_FILL`), or wireframes (`VK_POLYGON_MODE_LINE`).

```
VkPipelineRasterizationStateCreateInfo rasterizer{};
rasterizer.sType =
          VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
rasterizer.depthClampEnable = VK_FALSE;
rasterizer.rasterizerDiscardEnable = VK_FALSE;
rasterizer.polygonMode = VK_POLYGON_MODE_FILL;
rasterizer.lineWidth = 1.0f;
rasterizer.cullMode = VK_CULL_MODE_BACK_BIT;
rasterizer.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
rasterizer.depthBiasEnable = VK_FALSE;
rasterizer.depthBiasConstantFactor = 0.0f; // Optional
rasterizer.depthBiasClamp = 0.0f; // Optional
rasterizer.depthBiasSlopeFactor = 0.0f; // Optional
```

Back-face culling

Backface culling is controlled by the `cullMode` field: on with `VK_CULL_MODE_BACK_BIT`, and off with `VK_CULL_MODE_NONE`.

Accepted vertices order is contained into the `frontFace` field: either `VK_FRONT_FACE_CLOCKWISE` or `VK_FRONT_FACE_COUNTER_CLOCKWISE`.

```
VkPipelineRasterizationStateCreateInfo rasterizer{};
rasterizer.sType =
          VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
rasterizer.depthClampEnable = VK_FALSE;
rasterizer.rasterizerDiscardEnable = VK_FALSE;
rasterizer.polygonMode = VK_POLYGON_MODE_FILL;
rasterizer.lineWidth = 1.0f;
rasterizer.cullMode = VK_CULL_MODE_BACK_BIT;
rasterizer.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
rasterizer.depthBiasEnable = VK_FALSE;
rasterizer.depthBiasConstantFactor = 0.0f; // Optional
rasterizer.depthBiasClamp = 0.0f; // Optional
rasterizer.depthBiasSlopeFactor = 0.0f; // Optional
```

Z-buffer and stencil

Z-buffer and stencil are configured in a `VkPipelineDepthStencilStateCreateInfoStructure`.

```
VkPipelineDepthStencilStateCreateInfo depthStencil{};
depthStencil.sType =
          VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
depthStencil.depthTestEnable = VK_TRUE;
depthStencil.depthWriteEnable = VK_TRUE;
depthStencil.depthCompareOp = VK_COMPARE_OP_LESS;
depthStencil.depthBoundsTestEnable = VK_FALSE;
depthStencil.minDepthBounds = 0.0f; // Optional
depthStencil.maxDepthBounds = 1.0f; // Optional
depthStencil.stencilTestEnable = VK_FALSE;
depthStencil.front = {}; // Optional
depthStencil.back = {}; // Optional
```

Z-buffer and stencil

In particular, depth testing is enabled setting field `depthTestEnable` and `depthWriteEnabled` to `VK_TRUE`.

It is also possible to include or exclude the case in which two components have the same distance with the `depthCompareOp` field ("less than" is obtained with `VK_COMPARE_OP_LESS`).

```
VkPipelineDepthStencilStateCreateInfo depthStencil{};
depthStencil.sType =
          VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
depthStencil.depthTestEnable = VK_TRUE;
depthStencil.depthWriteEnable = VK_TRUE;
depthStencil.depthCompareOp = VK_COMPARE_OP_LESS;
depthStencil.depthBoundsTestEnable = VK_FALSE;
depthStencil.minDepthBounds = 0.0f; // Optional
depthStencil.maxDepthBounds = 1.0f; // Optional
depthStencil.stencilTestEnable = VK_FALSE;
depthStencil.front = {}; // Optional
depthStencil.back = {}; // Optional
```

Pipeline Layout and Uniforms Variables

The *pipeline layout* defines the uniform variables used by the shaders – we will describe this concept in-depth in a future lesson.

Even if no uniform variables are used, the section needs to be filled and the corresponding `VkPipelineLayout` object created (and destroyed at the end).

```
// Pipeline Layout (Uniform variables) - currently empty
VkPipelineLayout pipelineLayout;

VkPipelineLayoutCreateInfo pipelineLayoutInfo{};
pipelineLayoutInfo.sType =
     VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
pipelineLayoutInfo.setLayoutCount = 0; // Optional
pipelineLayoutInfo.pSetLayouts = nullptr; // Optional
pipelineLayoutInfo.pushConstantRangeCount = 0; // Optional
pipelineLayoutInfo.pPushConstantRanges = nullptr; // Optional

if (vkCreatePipelineLayout(device, &pipelineLayoutInfo, nullptr,
                &pipelineLayout) != VK_SUCCESS) {
     throw std::runtime_error("failed to create pipeline layout!");
}
```

Multisampling

Multisampling describes possible anti-aliasing techniques to reduce the artifacts and create smoother images. Also in this case, even if no anti-aliasing is used, the section needs to be filled with the following values specified inside a `VkPipelineMultisamplingStateCreateInfo` structure:

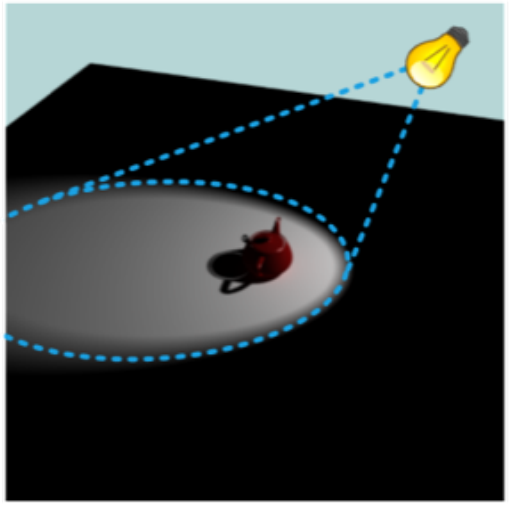
```
// Multisampling (in other things)
VkPipelineMultisampleStateCreateInfo multisampling{};
multisampling.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;

multisampling.sampleShadingEnable = VK_FALSE;
multisampling.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
multisampling.minSampleShading = 1.0f; // Optional
multisampling.pSampleMask = nullptr; // Optional
multisampling.alphaToCoverageEnable = VK_FALSE; // Optional
multisampling.alphaToOneEnable = VK_FALSE; // Optional
```

Render Passes

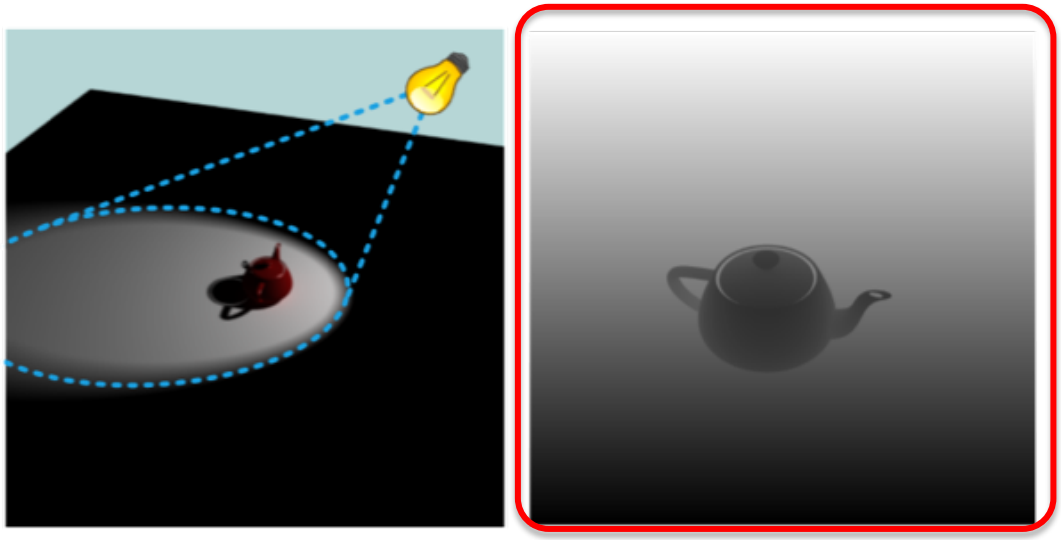
Rendering of current image might require several steps, and produce several images.

For example, to create shadows using the Shadow Map technique briefly introduced…



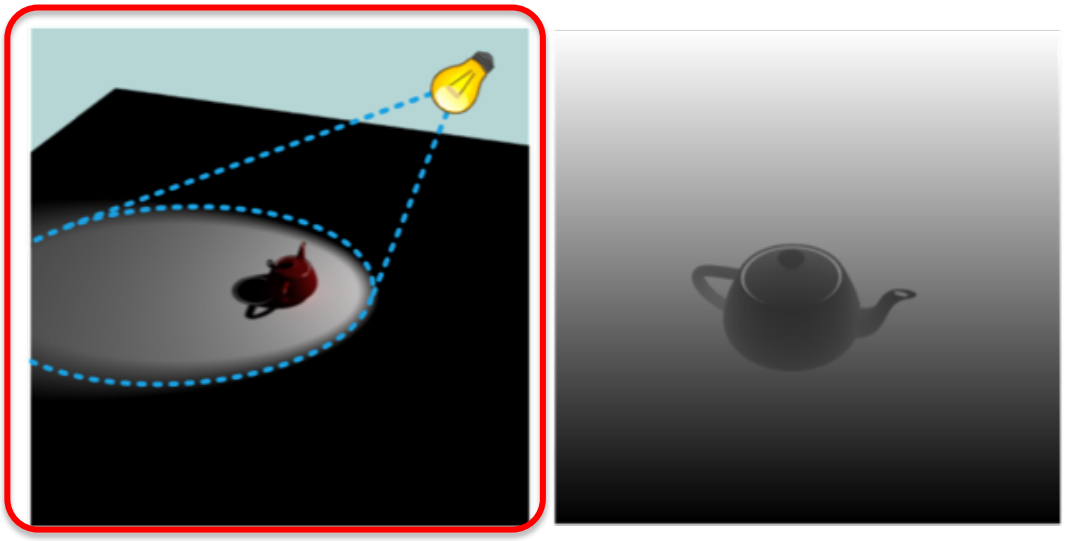
Render Passes

We need first to render the entire scene from the view of the light to produce a map the stores the distance of the points “seen” by the illumination source.



Render Passes

Then we need to render a second time the scene, this time using the map produced at the previous step to determine whether a point is directly hit by the light and perform the Ray-casting approximation to the rendering equation.



Render Passes

Render passes are defined inside a `VkRenderPass` object.

Their purpose is to contain inside a single data structure, all the information required for performing a similar algorithm.

```
// Render Pass configuration
VkRenderPass renderPass;
```

Color attachments

A render pass can produce a lot of information for every pixel, which can be used for advanced rendering techniques such as *Deferred Rendering*. These are stored in memory buffers corresponding to rectangular areas that are called *Attchments*. They are defined inside a `VkAttachmentDescription` object.

```
VkAttachmentDescription colorAttachment{};
    colorAttachment.format = surfaceFormat.format;
    colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
    colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
    colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
    colorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
    colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
    colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
    colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
```

Color attachments

Each attachment can have its own `format`, as described in the corresponding field. For conventional rendering techniques, this corresponds to the one defined for the drawing surface.

```
VkAttachmentDescription colorAttachment{};
    colorAttachment.format = surfaceFormat.format;
    colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
    colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
    colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
    colorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
    colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
    colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
    colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
```

Color attachments

Other parameters defines the operations that are allowed on the attachment, as well if stencils are used and possible conversions of the format done before or after its use.

A complete description of these fields is outside the scope of this course.

```
VkAttachmentDescription colorAttachment{};
    colorAttachment.format = surfaceFormat.format;
    colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
    colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
    colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
    colorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
    colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
    colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
    colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
```

Depth attachment

To support Z-buffering, the depth buffer must be defined into a specific attachment.

```
VkAttachmentDescription depthAttachment{};
depthAttachment.format = findDepthFormat(physicalDevice);
depthAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
depthAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
depthAttachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
depthAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
depthAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
depthAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
depthAttachment.finalLayout =
                    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```

Depth attachment format

Specific procedures can look for the best z-buffer pixel format.

As we did for surface creation, we will leverage the ones provided by the Vulkan tutorial.

```
VkFormat findDepthFormat(const VkPhysicalDevice &physicalDevice) {
return findSupportedFormat({VK_FORMAT_D32_SFLOAT, VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D24_UNORM_S8_UINT},
           VK_IMAGE_TILING_OPTIMAL, VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT, physicalDevice);
}

VkFormat findSupportedFormat(const std::vector<VkFormat> candidates,
VkImageTiling tiling, VkFormatFeatureFlags features, const VkPhysicalDevice physicalDevice) {
    for (VkFormat format : candidates) {
        VkFormatProperties props;
        vkGetPhysicalDeviceFormatProperties(physicalDevice, format, &props);
        if (tiling == VK_IMAGE_TILING_LINEAR && (props.linearTilingFeatures & features) == features) {
            return format;
        } else if (tiling == VK_IMAGE_TILING_OPTIMAL && (props.optimalTilingFeatures & features) == features) {
            return format;
        }
    }

    throw std::runtime_error("failed to find supported Z-buffer format!");
}
```

Attachment indices

Attachments are used through their id, wrapped inside `VkAttachmentReference` objects.

Their id is specified inside the `attachment` field, and their usage inside the `layout` property.

```
...
VkAttachmentReference colorAttachmentRef{};
colorAttachmentRef.attachment = 0;
colorAttachmentRef.layout =
          VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
...
VkAttachmentReference depthAttachmentRef{};
depthAttachmentRef.attachment = 1;
depthAttachmentRef.layout =
          VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```

Color Blending

For each color attachment (i.e. not the depth buffer), *color blending* must be defined. This controls whether new colors are combined with the ones previously present in the same position (and in this case how), or if they completely replace them.

```
VkPipelineColorBlendAttachmentState colorBlendAttachment{};
colorBlendAttachment.colorWriteMask =
          VK_COLOR_COMPONENT_R_BIT |
          VK_COLOR_COMPONENT_G_BIT |
          VK_COLOR_COMPONENT_B_BIT |
          VK_COLOR_COMPONENT_A_BIT;
colorBlendAttachment.blendEnable = VK_FALSE;
colorBlendAttachment.srcColorBlendFactor = VK_BLEND_FACTOR_ONE; // Optional
colorBlendAttachment.dstColorBlendFactor = VK_BLEND_FACTOR_ZERO; // Optional
colorBlendAttachment.colorBlendOp = VK_BLEND_OP_ADD; // Optional
colorBlendAttachment.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE; // Optional
colorBlendAttachment.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO; // Optional
colorBlendAttachment.alphaBlendOp = VK_BLEND_OP_ADD; // Optional
```

Color Blending

The information for all attachment is then collected inside a VkPipelineColorBlendStateCreateInfo structure, using its pAttachment field.

```
VkPipelineColorBlendStateCreateInfo colorBlending{};
colorBlending.sType =
     VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
colorBlending.logicOpEnable = VK_FALSE;
colorBlending.logicOp = VK_LOGIC_OP_COPY; // Optional
colorBlending.attachmentCount = 1;
colorBlending.pAttachments = &colorBlendAttachment;
colorBlending.blendConstants[0] = 0.0f; // Optional
colorBlending.blendConstants[1] = 0.0f; // Optional
colorBlending.blendConstants[2] = 0.0f; // Optional
colorBlending.blendConstants[3] = 0.0f; // Optional
```

Render Sub-passes

In advanced rendering techniques, frames must be rendered several times, each one possibly using different output buffers (i.e. attachments).

The execution of a complete rendering cycle is called Sub-pass and must be described with a `VkSubpassDescription` structure.

```
VkSubpassDescription subpass{};
subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
subpass.colorAttachmentCount = 1;
subpass.pColorAttachments = &colorAttachmentRef;
subpass.pDepthStencilAttachment = &depthAttachmentRef;

VkSubpassDependency dependency{};
dependency.srcSubpass = VK_SUBPASS_EXTERNAL;
dependency.dstSubpass = 0;
dependency.srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
dependency.srcAccessMask = 0;
dependency.dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
dependency.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
```

Render Sub-passes

Attachments references are inserted inside the `p…Attchment` fields:

```
VkSubpassDescription subpass{};
subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
subpass.colorAttachmentCount = 1;
subpass.pColorAttachments = &colorAttachmentRef;
subpass.pDepthStencilAttachment = &depthAttachmentRef;

VkSubpassDependency dependency{};
dependency.srcSubpass = VK_SUBPASS_EXTERNAL;
dependency.dstSubpass = 0;
dependency.srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
dependency.srcAccessMask = 0;
dependency.dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
dependency.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
```

Render Sub-passes

Synchronization between render sub passes can be define using the `VkSubpassDependency` objects.

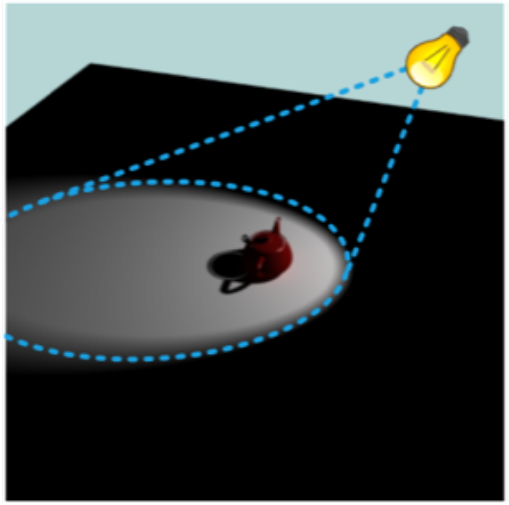
A complete description of this feature is however outside the scope of this course.

```
VkSubpassDescription subpass{};
subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
subpass.colorAttachmentCount = 1;
subpass.pColorAttachments = &colorAttachmentRef;
subpass.pDepthStencilAttachment = &depthAttachmentRef;

VkSubpassDependency dependency{};
dependency.srcSubpass = VK_SUBPASS_EXTERNAL;
dependency.dstSubpass = 0;
dependency.srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
dependency.srcAccessMask = 0;
dependency.dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
dependency.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
```

Render Passes: summary

To summarize, the considered *Shadow Map* example will require two color attachments (the shadow map and the image itself), and two sub-passes: one for creating the shadow map, and the second one to render the final image.



Render Pass creation

All the definitions about attachments and sub-passes are finally collected inside a `VkRenderPassCreateInfo`.

```
std::array<VkAttachmentDescription, 2> attachments =
                                {colorAttachment, depthAttachment};

VkRenderPassCreateInfo renderPassInfo{};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
renderPassInfo.attachmentCount = static_cast<uint32_t>(attachments.size());
renderPassInfo.pAttachments = attachments.data();
renderPassInfo.subpassCount = 1;
renderPassInfo.pSubpasses = &subpass;
renderPassInfo.dependencyCount = 1;
renderPassInfo.pDependencies = &dependency;
```

```
result = vkCreateRenderPass(device, &renderPassInfo, nullptr,
                &renderPass);
if (result != VK_SUCCESS) {
     throw std::runtime_error("failed to create render pass!");
}
```

Render Pass creation

The corresponding `VkRenderPass` object is finally created with the `vkCreateRenderPass()` function.

```
std::array<VkAttachmentDescription, 2> attachments =
                                {colorAttachment, depthAttachment};

VkRenderPassCreateInfo renderPassInfo{};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
renderPassInfo.attachmentCount = static_cast<uint32_t>(attachments.size());
renderPassInfo.pAttachments = attachments.data();
renderPassInfo.subpassCount = 1;
renderPassInfo.pSubpasses = &subpass;
renderPassInfo.dependencyCount = 1;
renderPassInfo.pDependencies = &dependency;

result = vkCreateRenderPass(device, &renderPassInfo, nullptr,
                &renderPass);
if (result != VK_SUCCESS) {
     throw std::runtime_error("failed to create render pass!");
}
```

Pipeline creation

The `VkPipeline` object is finally created by filling all the sections previously presented inside a `VkGraphicsPipelineCreateInfo` structure, using the `VkCreateGraphicsPipelines()` command.

Since an application might require several pipelines, the command can create an arbitrary number of them.

```
// Pipeline creation
VkGraphicsPipelineCreateInfo pipelineInfo{};
pipelineInfo.sType =
          VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
pipelineInfo.stageCount = 2;
pipelineInfo.pStages = shaderStages;
pipelineInfo.pVertexInputState = &vertexInputInfo;
pipelineInfo.pInputAssemblyState = &inputAssembly;
pipelineInfo.pViewportState = &viewportState;
pipelineInfo.pRasterizationState = &rasterizer;
pipelineInfo.pMultisampleState = &multisampling;
pipelineInfo.pDepthStencilState = &depthStencil;
pipelineInfo.pColorBlendState = &colorBlending;
pipelineInfo.pDynamicState = nullptr; // Optional
pipelineInfo.layout = pipelineLayout;
pipelineInfo.renderPass = renderPass;
pipelineInfo.subpass = 0;
pipelineInfo.basePipelineHandle = VK_NULL_HANDLE;
pipelineInfo.basePipelineIndex = -1; // Optional

result = vkCreateGraphicsPipelines(device, VK_NULL_HANDLE,
          1, &pipelineInfo, nullptr, &Pipeline);
if (result != VK_SUCCESS) {
     throw std::runtime_error(
               "failed to create graphics pipeline!");
}
```

Pipeline destruction

When the pipeline is no longer needed, all resources (including the pipeline itself, its layout and the render passes), must be released with the following commands:

```
vkDestroyPipeline(device, Pipeline, nullptr);
vkDestroyPipelineLayout(device, pipelineLayout, nullptr);
vkDestroyRenderPass(device, renderPass, nullptr);
```