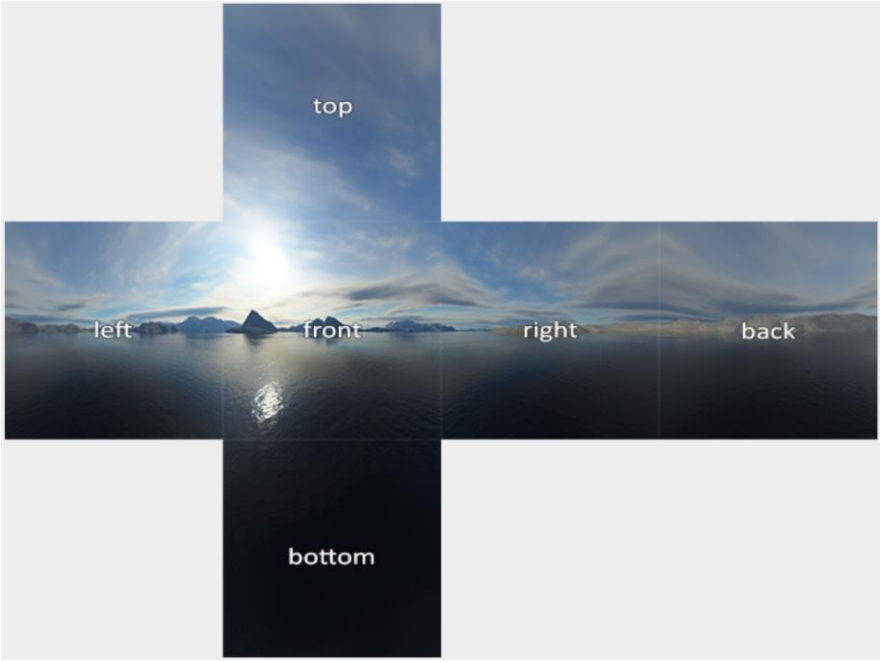
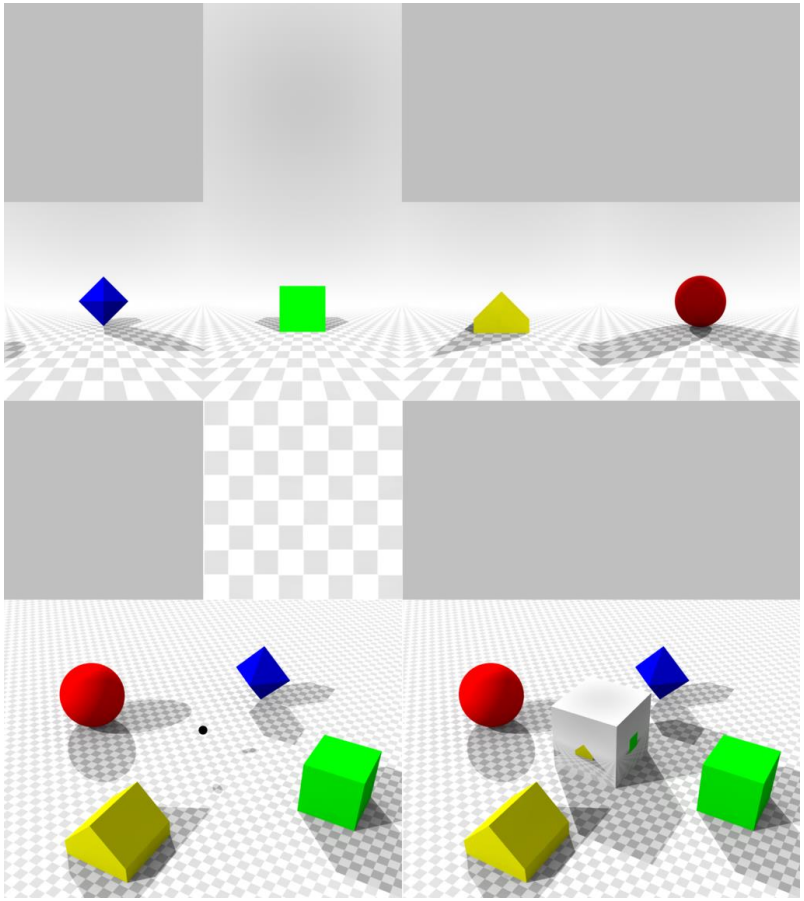
Textures and Samplers

Cube map textures

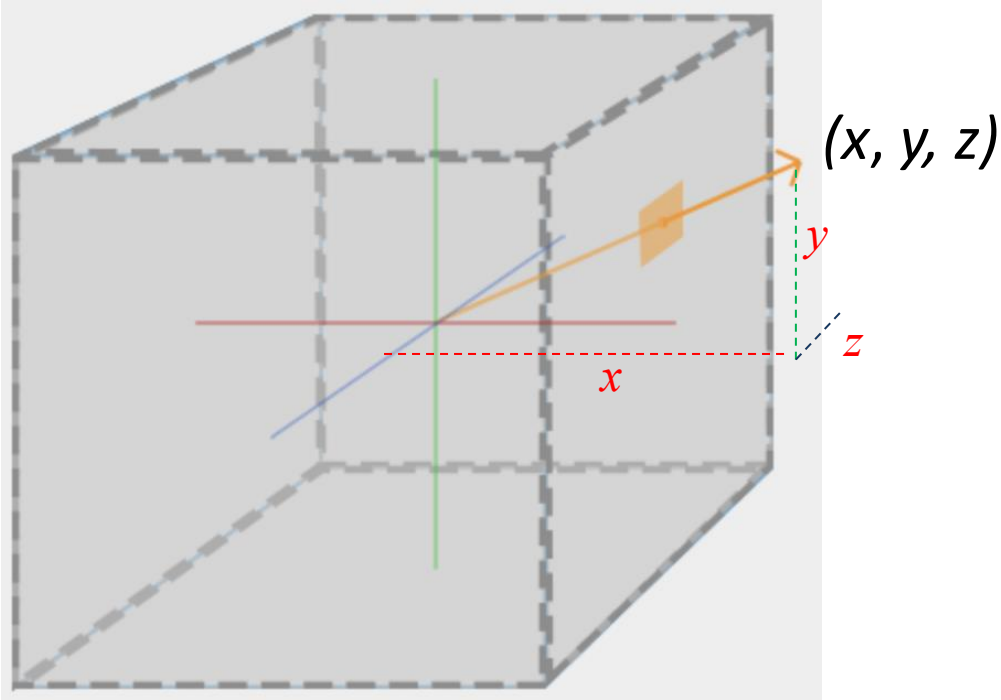
Cube-map textures are collections of six images, that correspond to the six sides of a cube.





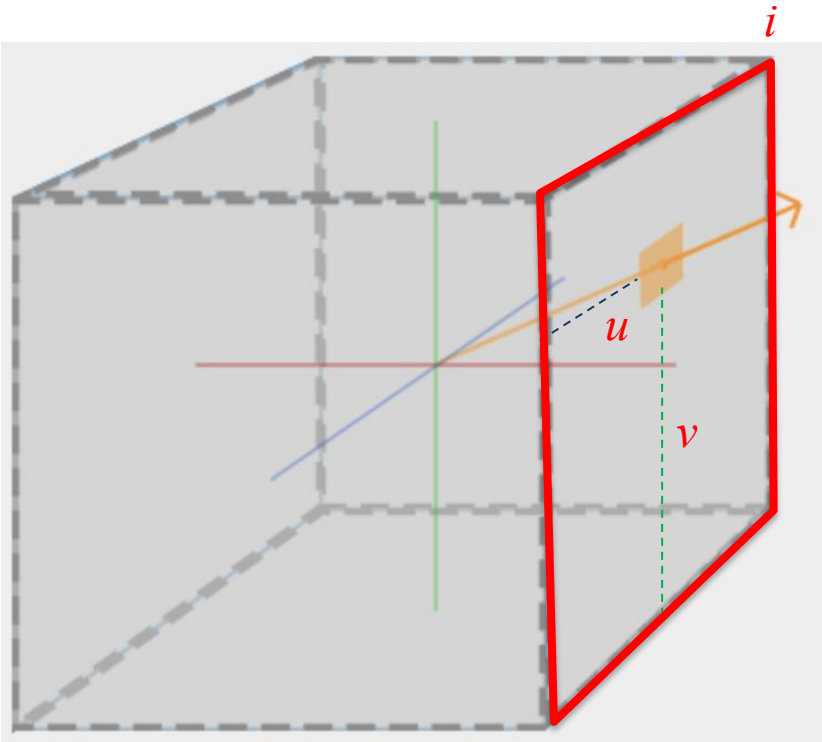
Cube map textures

They are indexed by a direction vector, specified with its *x*, *y* and *z* components.



Cube map textures

These coordinates determine both the side of the cube (the base image), and the pixel on such image to fetch.



Cube map textures

In modern applications such textures are particularly important since they are used as building blocks to create realistic rendering effects that emulates mirror reflection and transparency.

We will briefly return on this topic later in the course.

UV intervals

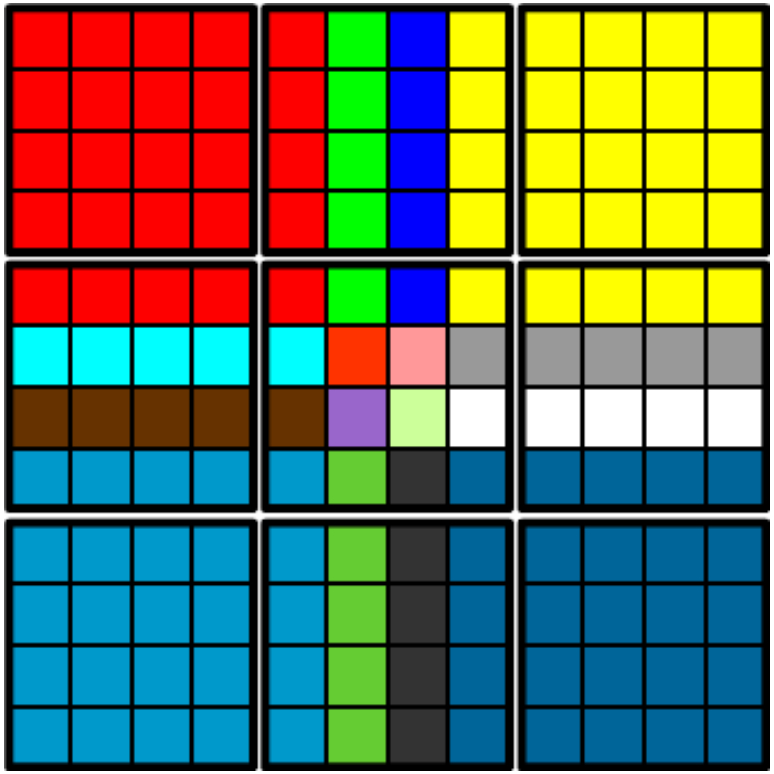
In some cases, the *u* or *v* values can fall outside the *[0, 1*] range.

Several behaviors are possible, and the four most popular are:

- *Clamp*
- *Repeat*
- *Mirror*
- *Constant*

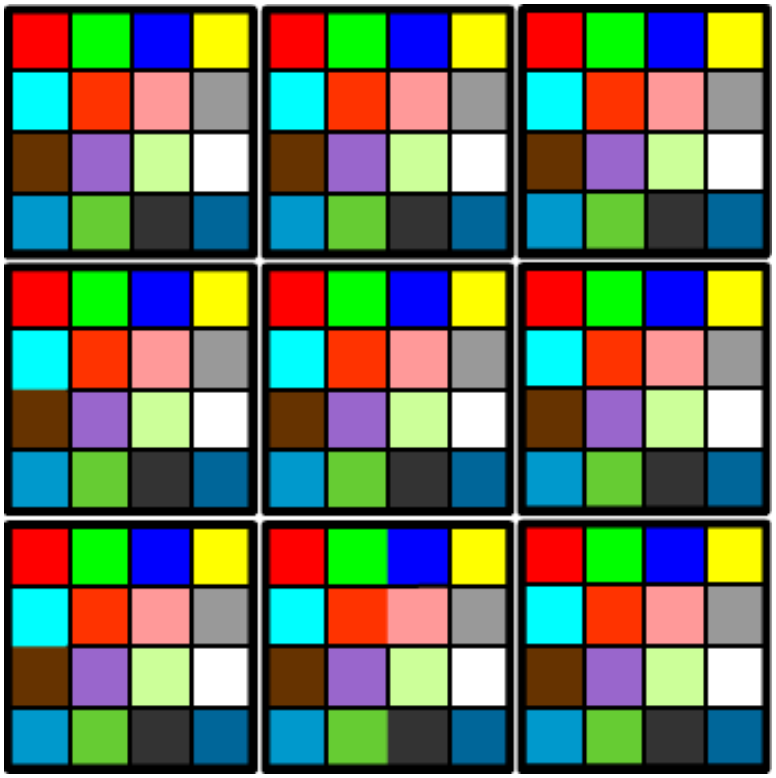
Clamp

Clamp extends the colors of the border to the texels that are outside the *[0,1]* range.



Repeat

The *repeat* strategy, considers only the fractional part of the UV coordinates, and discards their integer part.

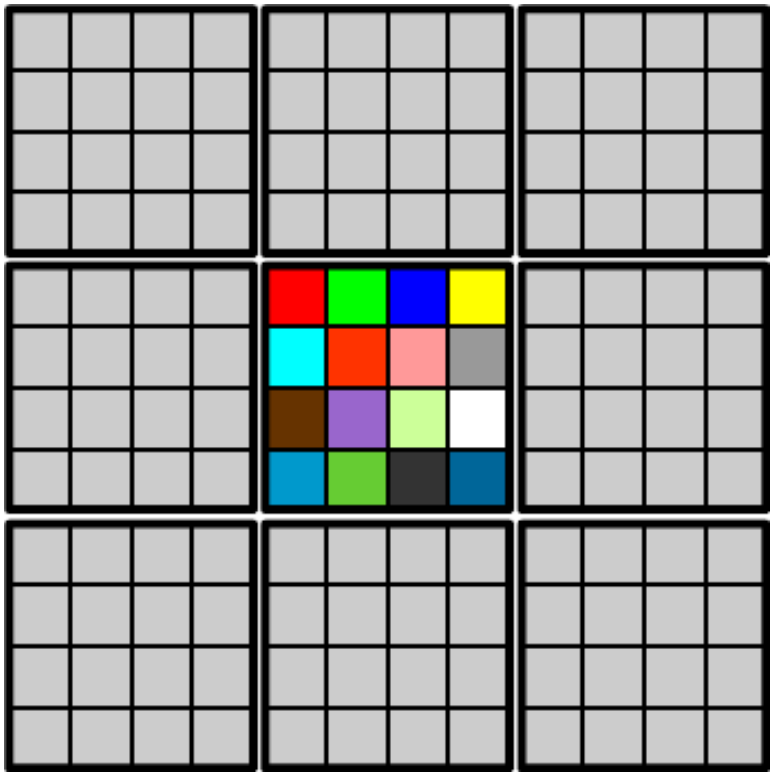


Mirror

The *mirror* technique, repeats the pattern of the texture, but it flips at every repetition.

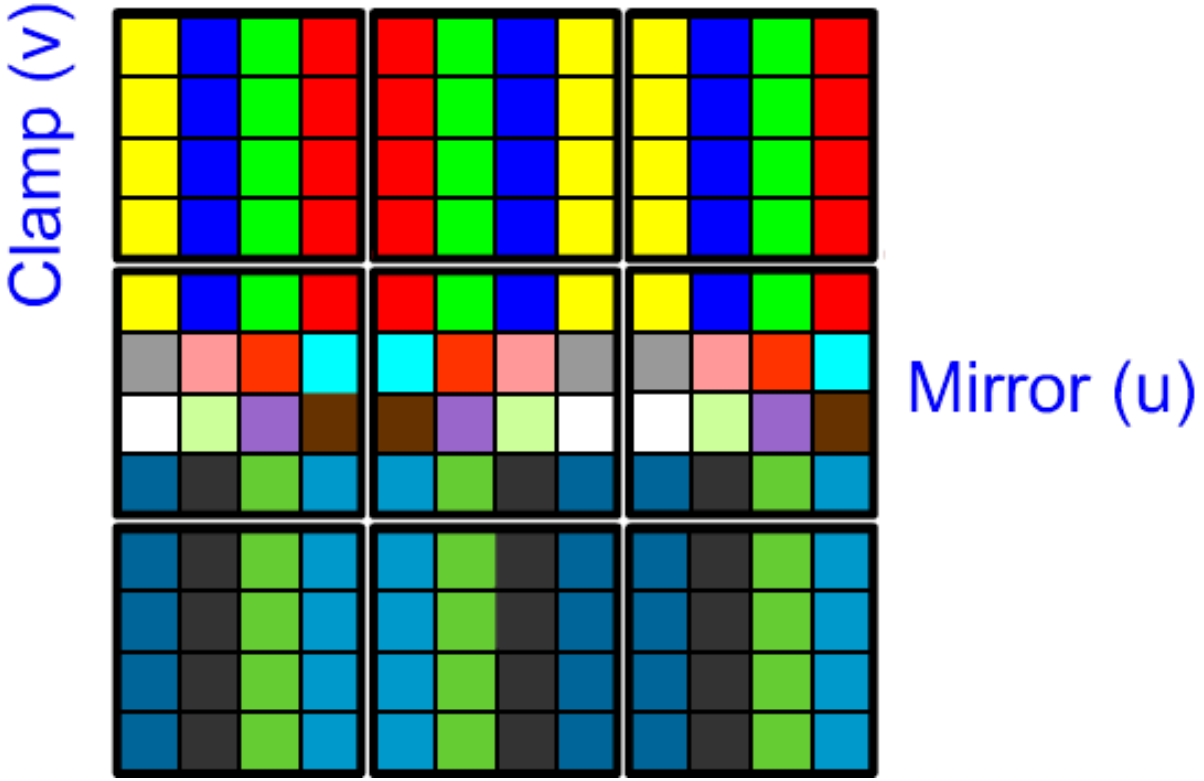
Constant

The constant behavior replaces the samples of the texture that fall outside the *[0, 1]* range with a default color c_d.



UV intervals

By combining different strategies for *u* and *v*, several effects can be obtained.



Texture filtering

UV coordinates are floating point numbers, while texels are indexed by integer values.

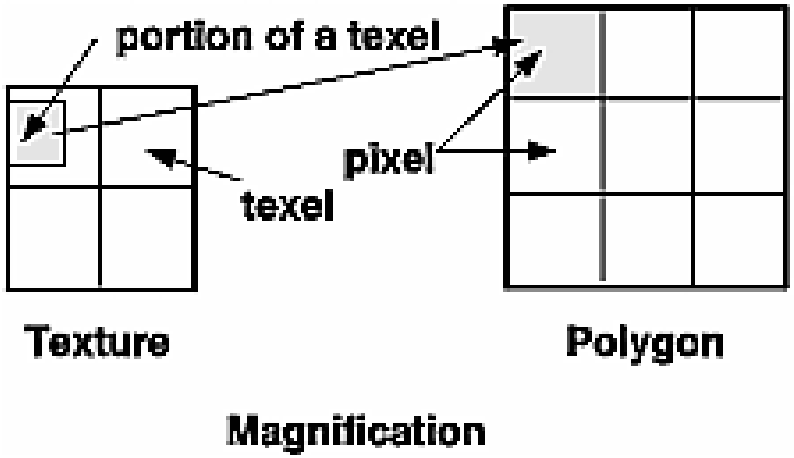
Moreover, the shape of a pixel in the screen might correspond to several texels on the texture.

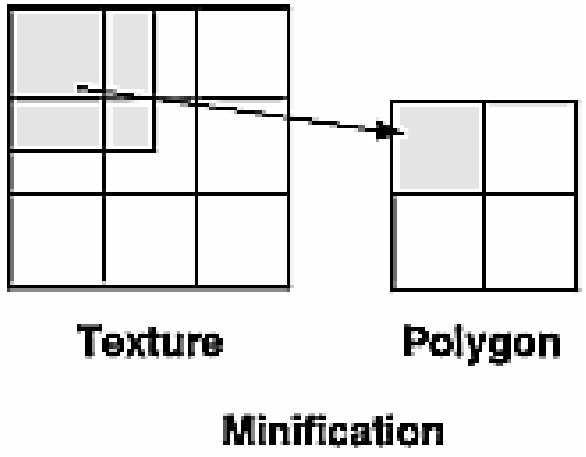
Such problems are solved using techniques known as *filtering*.

Texture filtering

When a texel is larger than the corresponding pixel, we have a *magnification filtering* problem.

When the pixel on screen corresponds to several texels, we have a *minification filtering* problem. This second case is much harder to deal with than the first.





Magnification filters

Two magnification filters are usually defined:

- *Nearest pixel*
- *(bi)linear interpolation*

Nearest pixel

Let us consider a texture of size *w* x *h.* Texels over the *u* axis are indexed from *0* to *w-1*, and over the *v* axis from *0* to *h-1*. Texel *p[0][0]* is at the one corner of the image.

With the *nearest pixel* filter, the lookup procedure first transforms the UV coordinates with respect to the texture size, then returns the texel *p[i][j]* that considers only the integer part of the scaled coordinates.

$$x = u \cdot w \qquad i = \lfloor x \rfloor$$

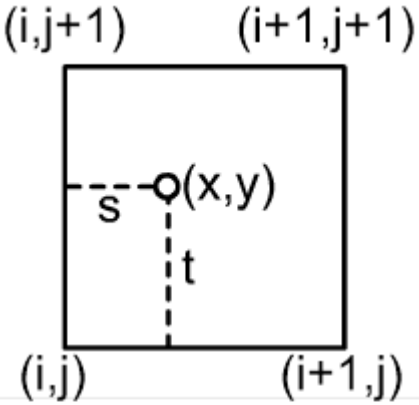
$$y = v \cdot h \qquad j = \lfloor y \rfloor$$

Nearest pixel

It is very fast, and requires just reading one texel per pixel, but it produces blocky images.

(Bi)linear interpolation

Linear interpolation interpolates the color of the pixel from the values of its closest neighbors.



$$
\begin{aligned}
p' \quad = \quad & (1-s)\,(1-t)\,p_{i,j} + s\,(1-t)\,p_{i+1,j} + \\
& +(1-s)\,t\,p_{i,j+1} + s\,t\,p_{i+1,j+1}
\end{aligned}
$$

(Bi)linear interpolation

The technique produces smooth results, but it requires *4 texture accesses* and *3 interpolations*.

(Bi)linear interpolation

Although the technique is usually called bilinear interpolation, the name is only appropriate for 2D textures.

For 1D textures, it requires just the interpolation over one axis (2 texels, 1 interpolation), and for 3D textures it requires interpolation over three dimensions (8 texels, 7 interpolations).

For this reason the name "bilinear" has been dropped and replaced simply with "linear" interpolation. The original name is however still very commonly used.

Minification filters

The most common minification filters are:

- *Nearest pixel*
- *(bi)linear interpolation*
- *Mip-mapping*
- *(Tri)linear interpolation*
- *Rip-mapping*
- *Anisotropic*

Nearest pixel and bilinear interpolation perform exactly in the same way as for the magnification: however they both give very poor results if the reduction is large.

This is due to the fact that minification should average the texels that fall inside a pixel, instead of simply guessing an intermediate value.

Mip-mapping

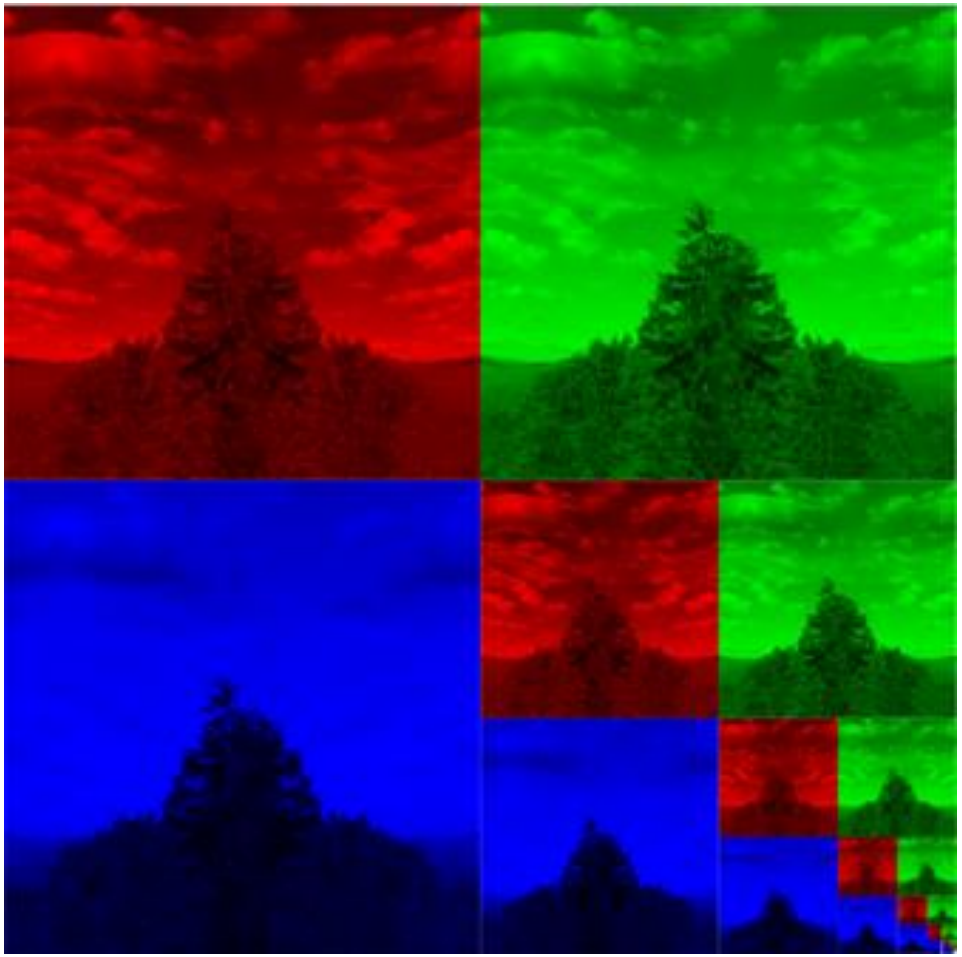
The easiest solution is called *Mip-mapping*. (*MIP = Multum In Parvo*).

The technique pre-computes a set of scaled versions of the texture (called *levels*), each one with halved size, and fetches the texel from the one whose size is closest to the one of the pixel on screen.

Mip-mapping

A Mip-map requires *33%* extra space, and can be generated on the fly when the texture is loaded, or manually created off-line.

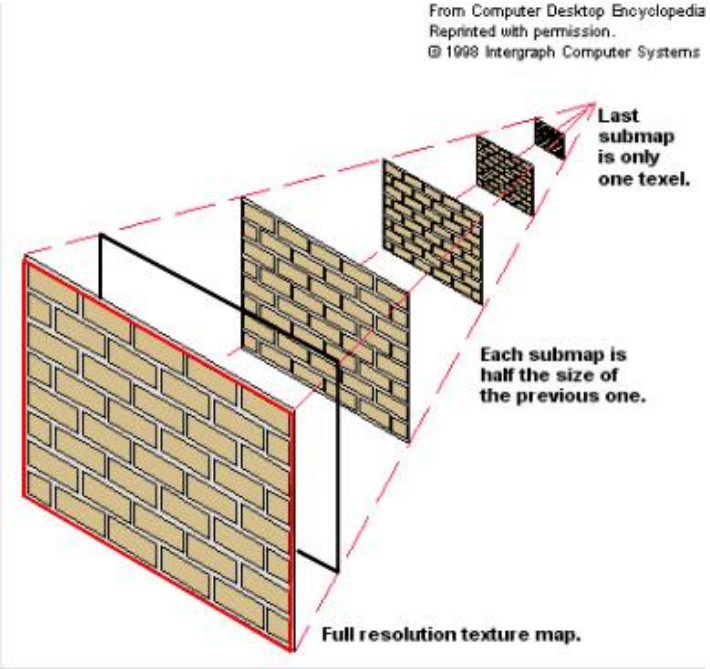
In Vulkan, this process must be manually programmed by the developer: we will return on this later.



Mip-mapping

Each pixel of an inner level of a Mip-map corresponds to the integration of a set of pixels in the original image.

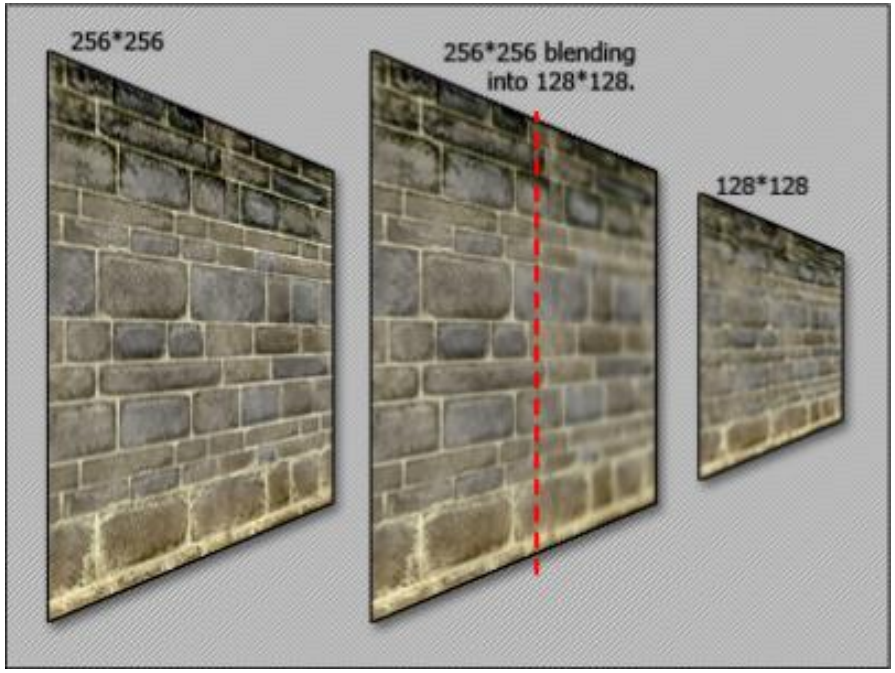
Depending on the ratio between the texture and the pixels, the algorithm selects the closest image in the mip-map.



Trilinear filtering

When images are angled with respect to the viewer, there might be a change of level inside their mapping.

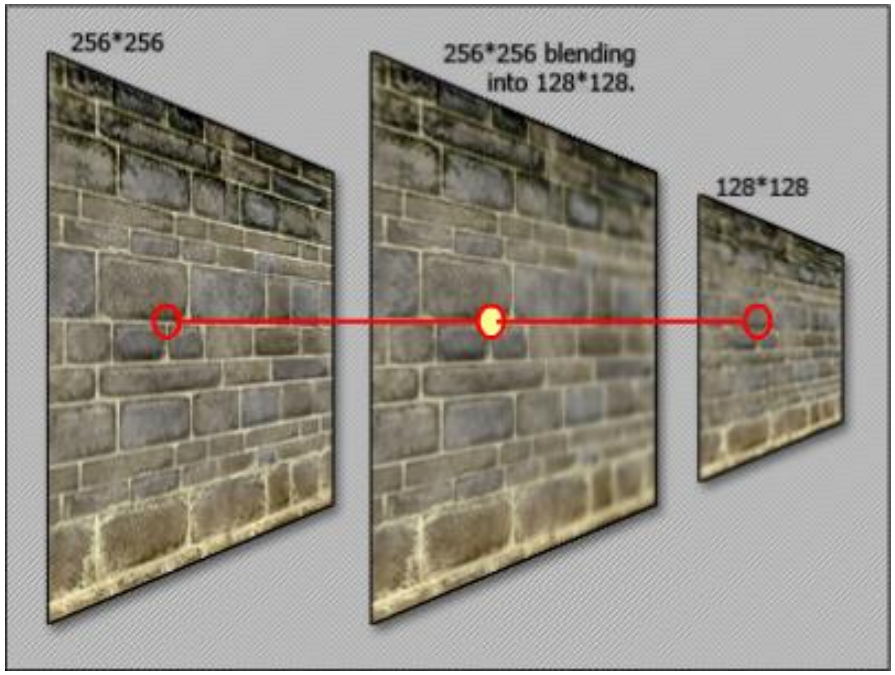
This might cause visible artifacts due to the abrupt change.



Trilinear filtering

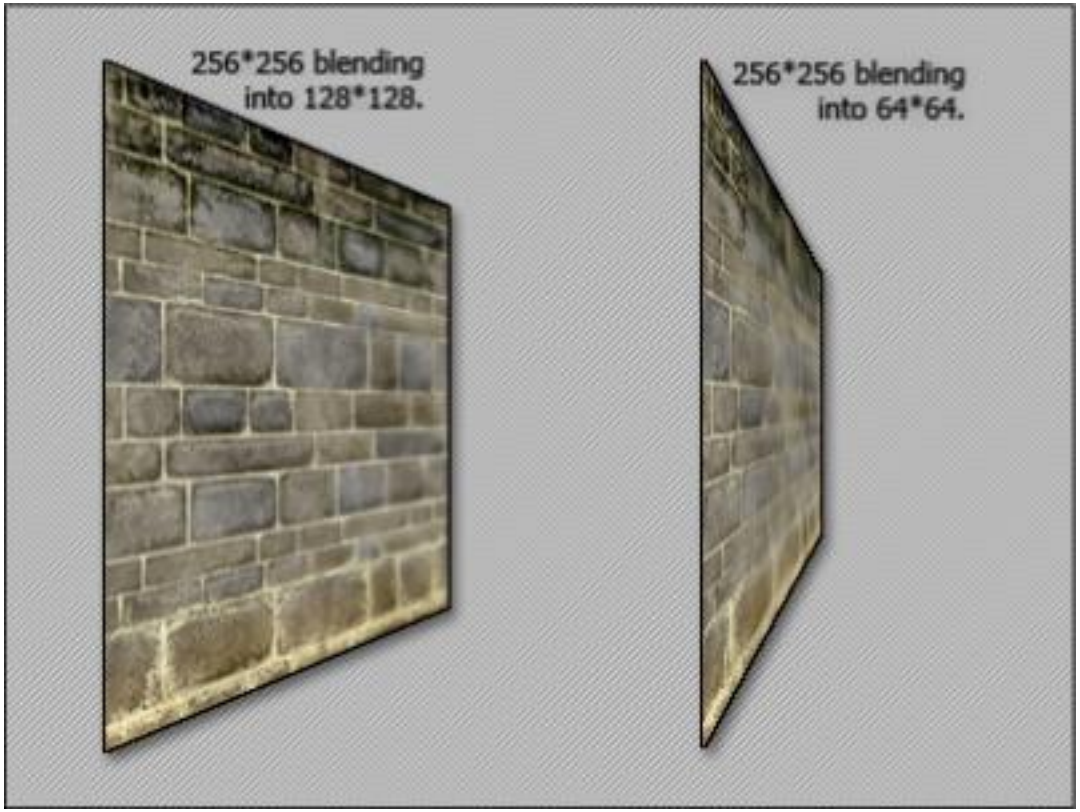
(Tri)linear filtering computes the required pixel two times, using the two closest levels of the Mip-map.

Then interpolates the two pixels, according to the difference between the texel and the corresponding pixel.



Trilinear filtering

In this way a smooth transition among the Mip-map levels is created.

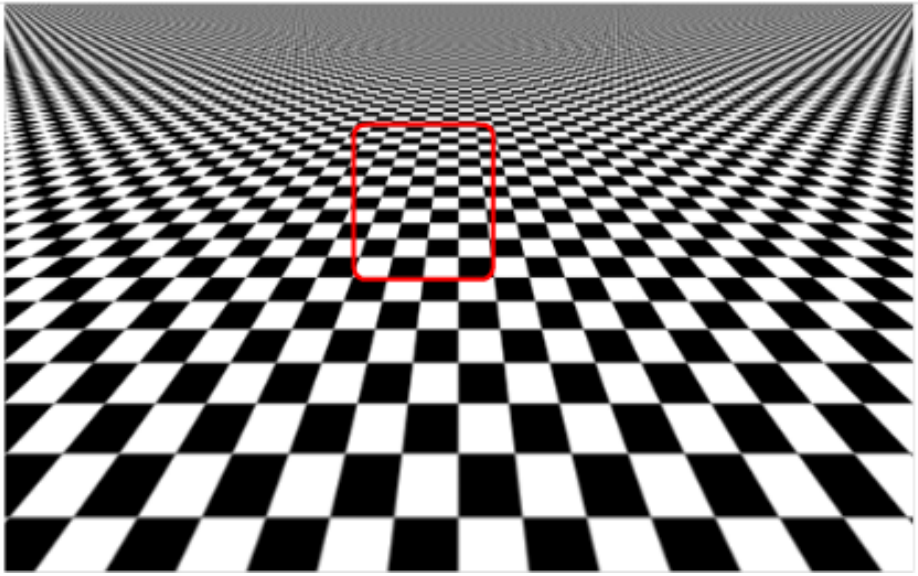


Rip-mapping

The Mip-map approach tends to produce blurred images when surfaces are angled with respect to the viewer.

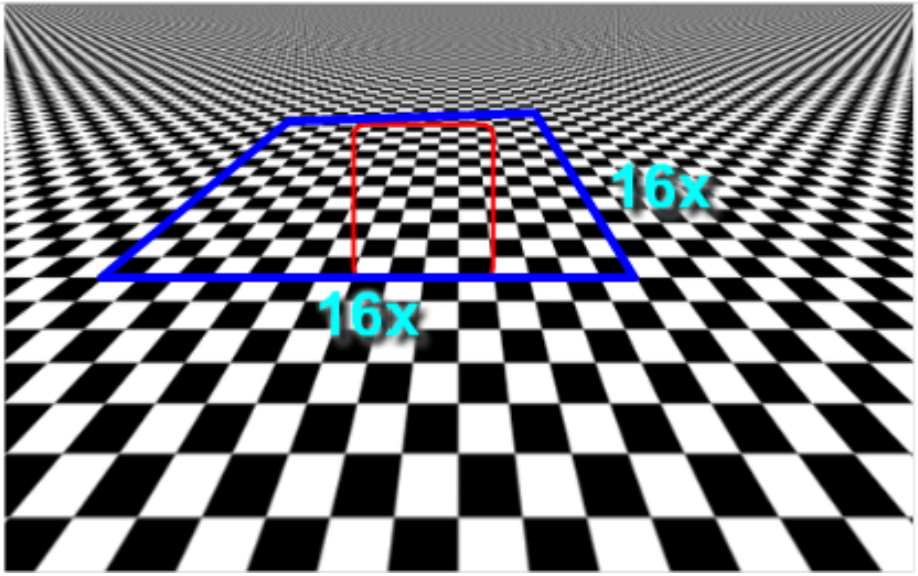
Rip-mapping

This happens because in this case pixels on screen corresponds to rectangles on the texture, and the Mip-map considers only square regions.



Rip-mapping

The texel of the Mip-map can be effectively used only for one direction of the rectangle.

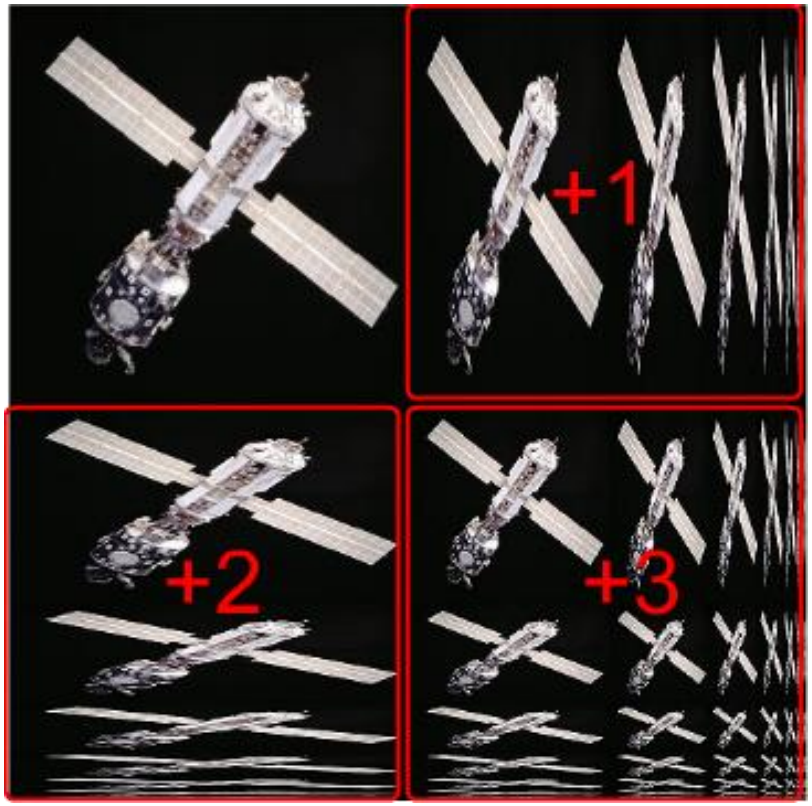


Rip-mapping

A *Rip-map* (Rip = Rectim in Parvo) encodes also rectangular scaling of the original texture.

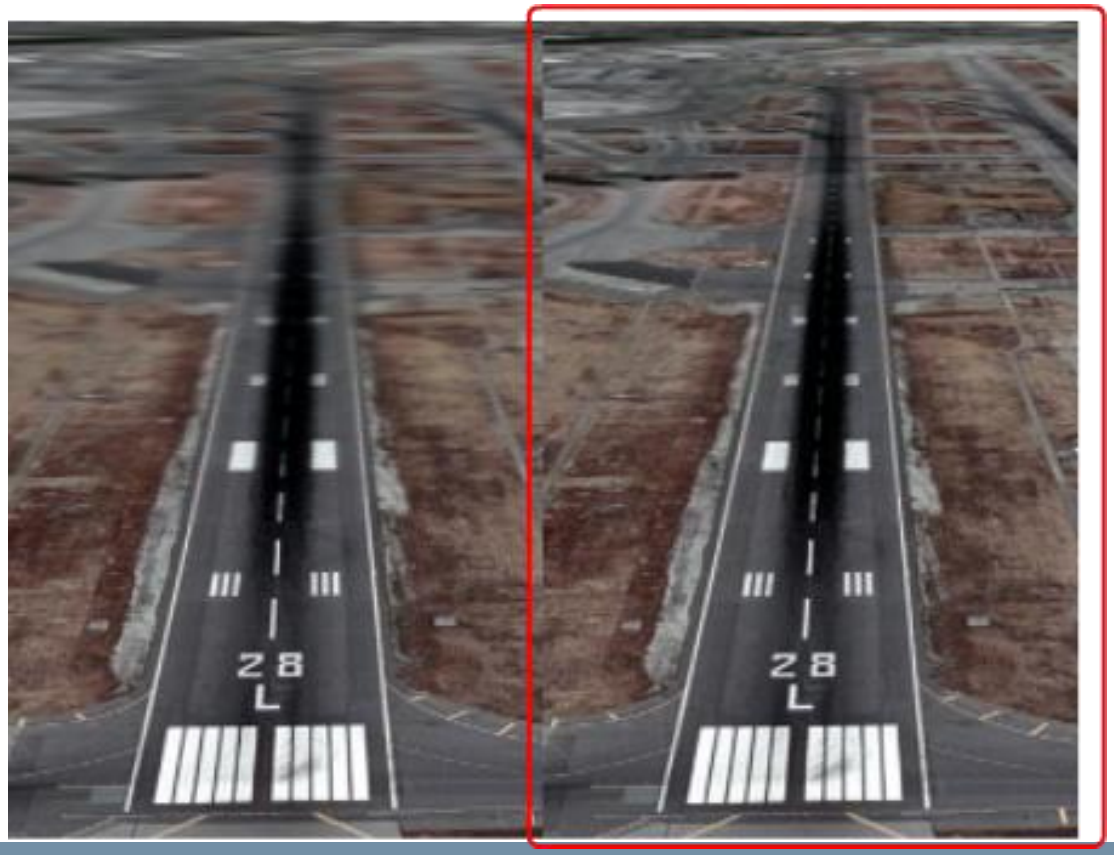
Rip-mapping

A Rip-map requires 4x the memory of a standard texture, it can be either pre-computed or generated on the fly.



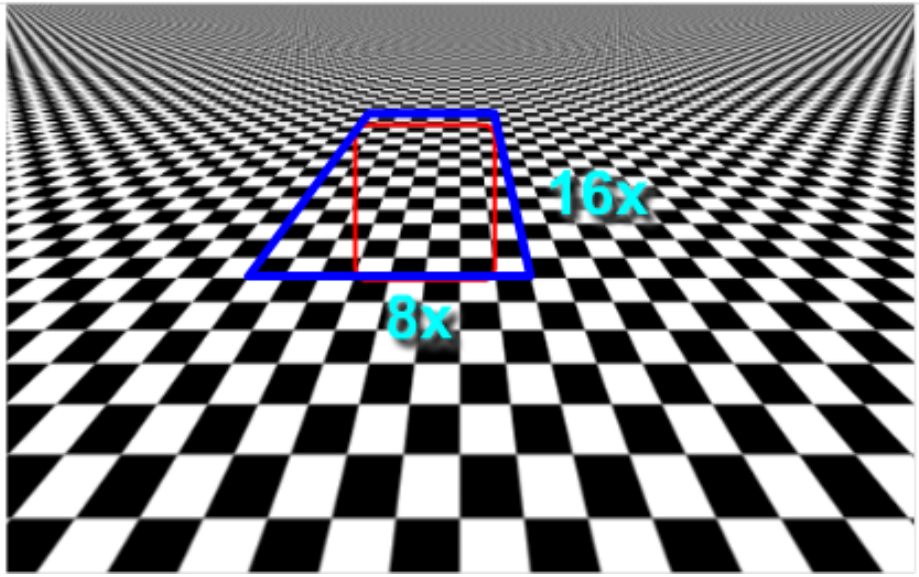
Rip-mapping

Using the rectangular version of the map, the algorithm can provide sharp results in (some) angled views.



Rip-mapping

In this case the algorithm can choose texels that corresponds to rectangular areas of the original texture, allowing it to obtain better performances for the cases in which the texture is aligned with camera.

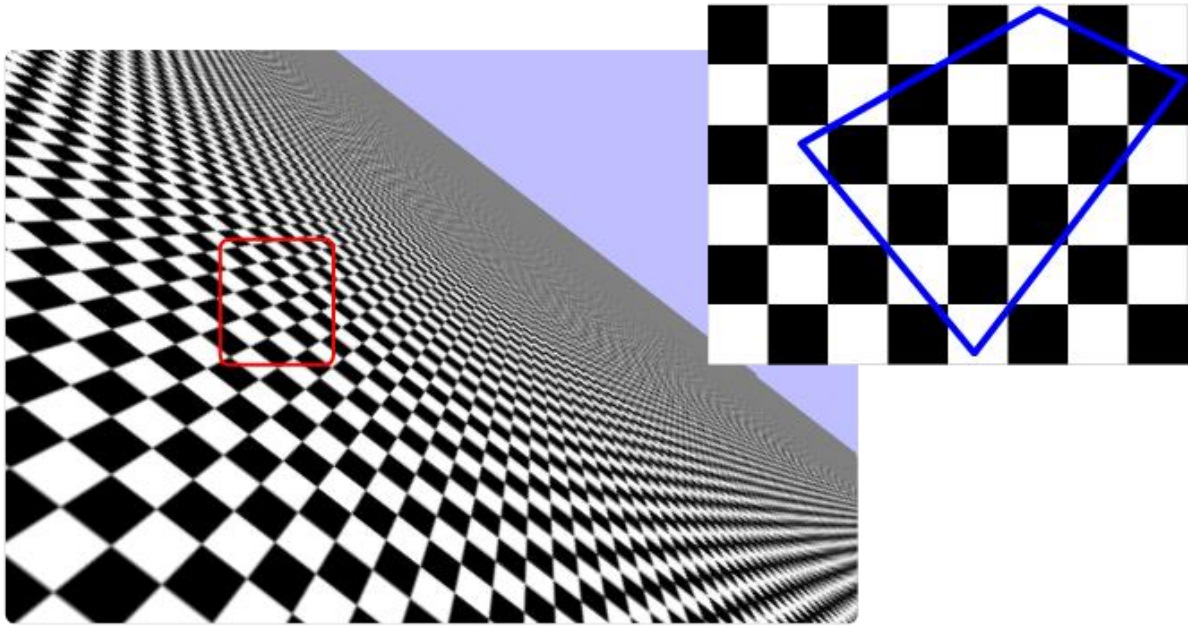


Although theoretically interesting, the Rip-mapping technique is basically never implemented due to its large memory requirements and its inability to deal with angled planes.

Anisotropic

Rip-mapping is not able to deal with surfaces angled with respect to the border of the screen.

In this case the pixel on screen corresponds to a generic trapezoid over the texture (that is, neither a square, nor a rectangle).



Anisotropic

A solution that can deal with angled surfaces is the *Anisotropic Unconstrained Filter*.

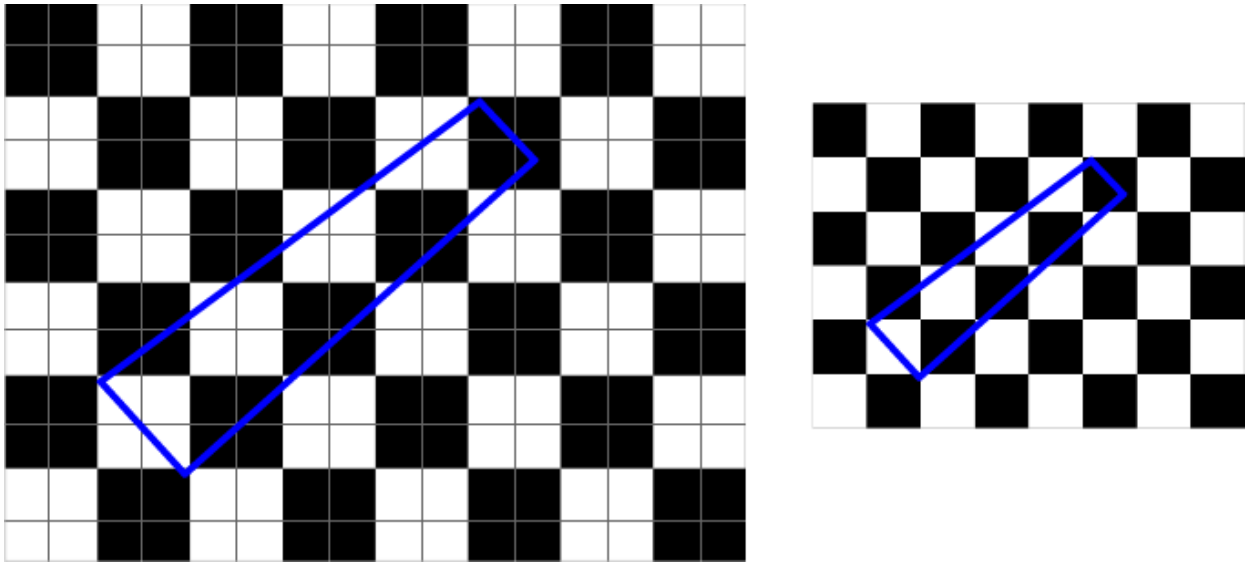
The algorithm tries to approximate the exact area of texels that corresponds to a pixel on screen.

It uses Mip-maps, and samples them following the direction of the pixel over the texels.

Anisotropic

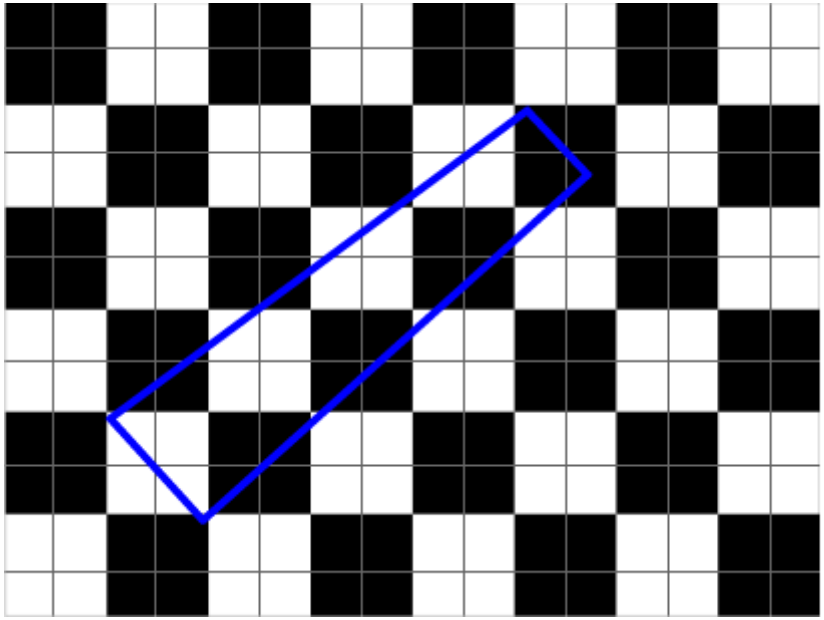
The algorithm first determines the position on the texture of the four borders of the pixel.

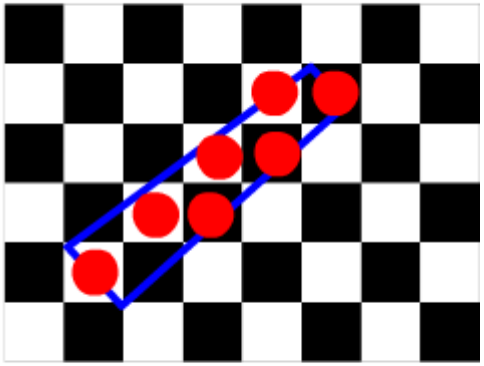
Depending on the size of the smallest side of the trapezoid, the procedure determines the most appropriate Mip-map level (the one with the most similar pixel to texel ratio).



Anisotropic

Then it samples the selected Mip-map level on a line in the direction of the on-screen pixel.





Anisotropic

The number of samples is generally limited to a maximum value (usually 8 or 16).

Although very effective, the method has a large overhead that can slow down rendering.

Vulkan allows to enable the Anisotropic filtering if supported by the graphic adapter: we will investigate this opportunity in a future lesson.

Textures in Shaders

Textures are passed to shaders as particular uniform variables of “Combined Texture Sample” type.

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

layout(location = 0) out vec4 outColor;

layout(binding = 1) uniform sampler2D texSampler;


void main() {
     vec3 Diffuse = texture(texSampler, fragUV).rgb ;

     outColor = vec4(Diffuse, 1.0);

}
```

Textures in Shaders

A lot of different samplers exists. The most important ones are:

- `sampler1D`
- `sampler2D`
- `sampler3D`
- `samplerCube`

Textures in Shaders

The shader, can obtain the color at a given position with the `texture()` command.

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

layout(location = 0) out vec4 outColor;

layout(binding = 1) uniform sampler2D texSampler;


void main() {
     vec3 Diffuse = texture(texSampler, fragUV).rgb ;

     outColor = vec4(Diffuse, 1.0);

}
```

Textures in Shaders

The first parameter determines the images, and the second the where to read the texel (with a format dependent on the sampler type).

The function returns a `vec4` color, where the last component is alpha channel (transparency).

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

layout(location = 0) out vec4 outColor;

layout(binding = 1) uniform sampler2D texSampler;


void main() {
     vec3 Diffuse = texture(texSampler, fragUV).rgb ;

     outColor = vec4(Diffuse, 1.0);

}
```

Textures in Shaders

For Mip-mapped images, we can use the `textureLod()` command to read from an image of a specified size.

```
textureLod(sampler, pos, lod)
```

The `sample` and `pos` parameters are identical to the conventional version of the command.

The `lod` parameter addresses the image with the highest detail with *0*, the first reduction with *1*, the second reduction with *2*, and so on up to the number of available levels..

Separate Texture and Sampler

Vulkan also allows to specify the texture and its sampler in two different uniforms.

We consider this opportunity outside the scope of this course, and we will not investigate it.

19 lines (17 sloc) | 721 Bytes

```
#version 400
#extension GL_ARB_separate_shader_objects : enable
#extension GL_ARB_shading_language_420pack : enable
layout (set = 0, binding = 1) uniform texture2D tex;
layout (set = 0, binding = 2) uniform sampler samp;
layout (location = 0) in vec2 inTexCoords;
layout (location = 0) out vec4 outColor;
void main() {

    // Combine the selected texture with sampler as a parameter
    vec4 resColor = texture(sampler2D(tex, samp), inTexCoords);

    // Create a border to see the cube more easily
   if (inTexCoords.x < 0.01 || inTexCoords.x > 0.99)
       resColor *= vec4(0.1, 0.1, 0.1, 1.0);
   if (inTexCoords.y < 0.01 || inTexCoords.y > 0.99)
       resColor *= vec4(0.1, 0.1, 0.1, 1.0);
   outColor = resColor;
}
```

UV transforming

UV coordinates can be transformed before performing a texture lookup.

This allows scaling or rotating a texture before applying it to a surface.

If the transformation is animated, this can produce interesting dynamic effects (e.g. a rotating fan).

UV transforming

In the simplest case, a *(u, v)* coordinate is inserted into a homogeneous coordinate *(u, v, 0, 1)*.

The homogeneous coordinate is then transformed with a texture coordinate transformation matrix M_t (created in the same way as the *World transform*).

The result is then transformed back into UV coordinates by simply discarding the last two elements of the four component vector.

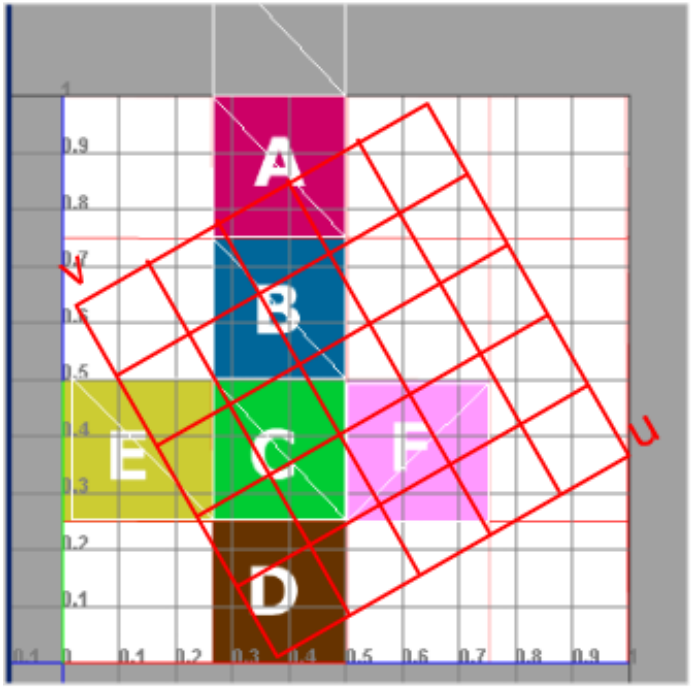
$$\begin{vmatrix} u & v \end{vmatrix} \rightarrow t_c = \begin{vmatrix} u & v & 0 & 1 \end{vmatrix}$$

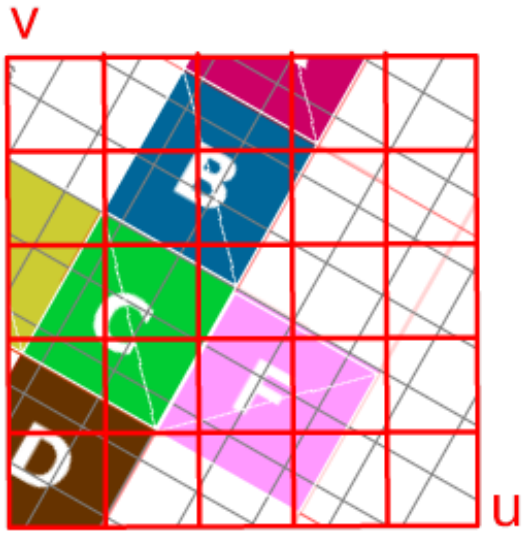
$$t'_c = M_t \cdot t_c$$

$$t'_c = \begin{vmatrix} u' & v' & 0 & 1 \end{vmatrix} \rightarrow \begin{vmatrix} u' & v' \end{vmatrix}$$

UV transforming

In this way the original texture can be rotated or scaled before being applied to the object.





UV transforming

Note that even if the UV coordinates associated to a vertex are in the range [0,1], they can become negative or greater than one after the transform.

Thus, when UV coordinates are transformed, the UV clamping techniques become of paramount importance.