POLITECNICO
MILANO 1863

DIPARTIMENTO DI ELETTRONICA
INFORMAZIONE E BIOINGEGNERIA

DE
IB

2024

Dipartimento di Elettronica, Informazione e Bioingegneria

Computer Graphics

Milano, 2024

Computer Graphics

- Textures and UV Mapping

POLITECNICO MILANO 1863

Texture

The appearance of realistic 3D objects should be defined in great detail, 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.

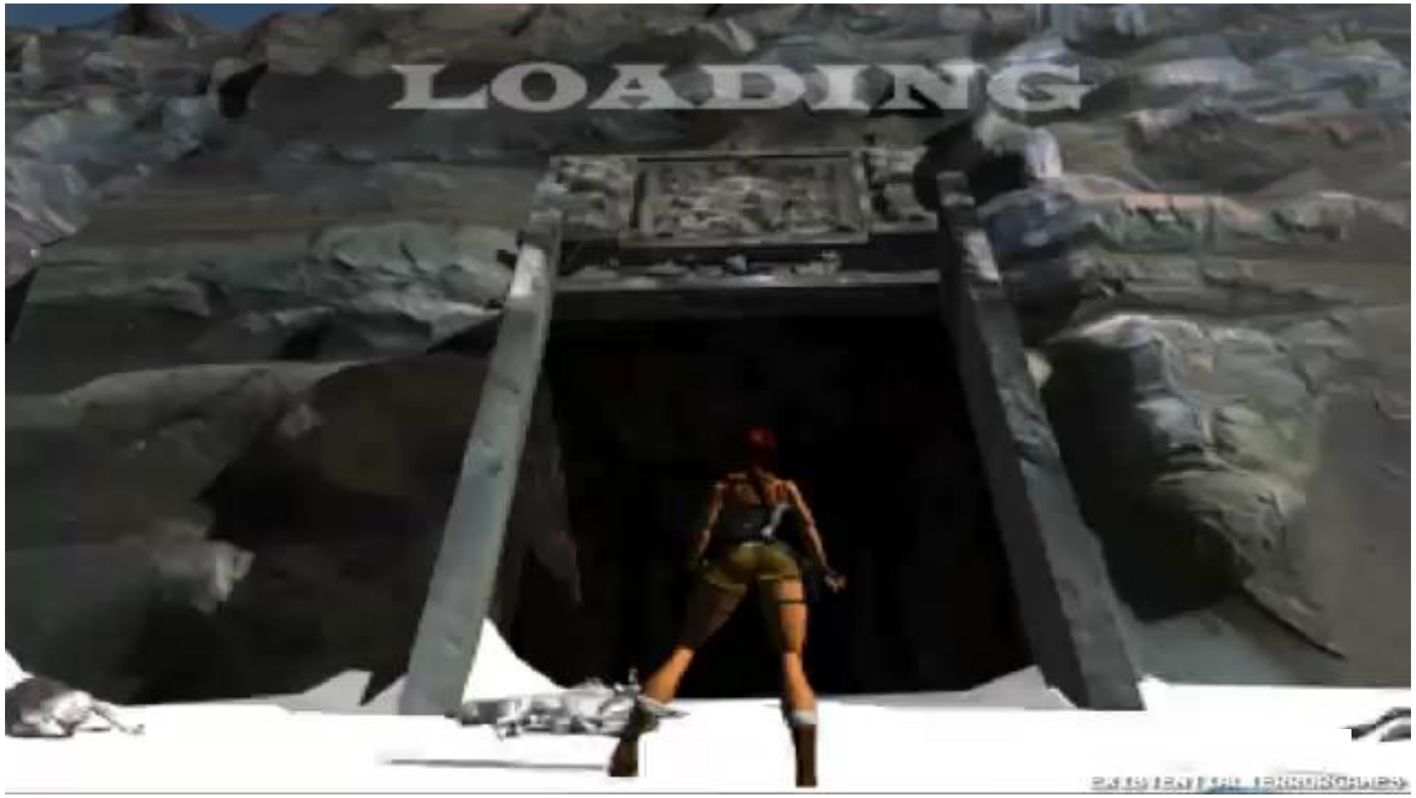
The common approach is to use tables that assign a different value to the parameters of the shaders, depending on the positions of the internal points of the surface.

In the most of the common cases, such tables contain the diffuse color of the object, which is acquired from an image. However, these tables are also used to store other properties that can parametrize the BRDF of a surface.

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

Texture

Example of use of texture effects can be seen in the 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 maps

Texture

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

3D textures also define parameters for a volume.

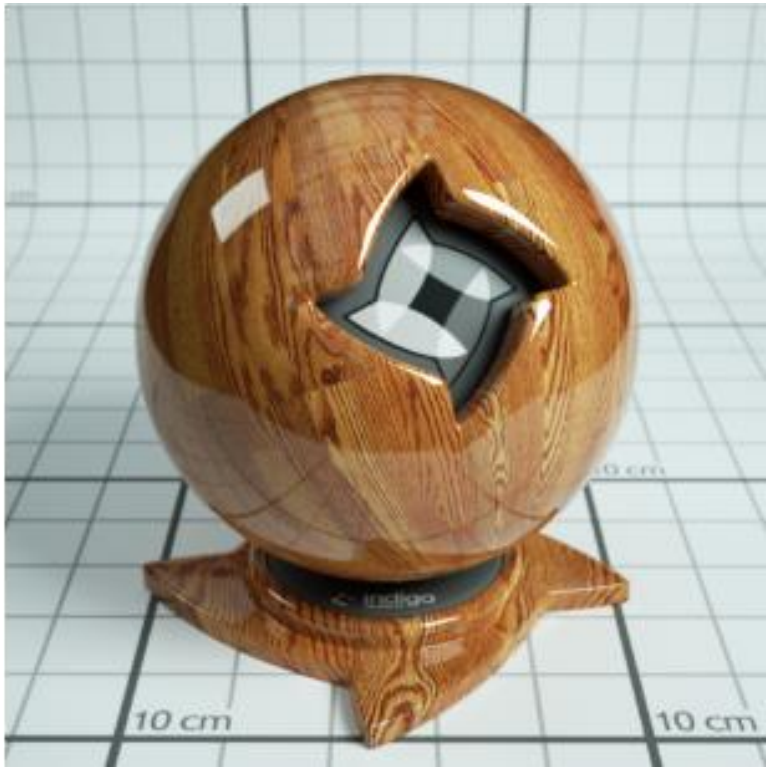
1D textures are instead used to contain values of pre-computed functions, or to perform optimizations.

Cube maps textures are instead used to define parameters associated with *directions* in a 3D coordinate system and they use a slightly different approach.

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 3D topologies, composed of several sides.

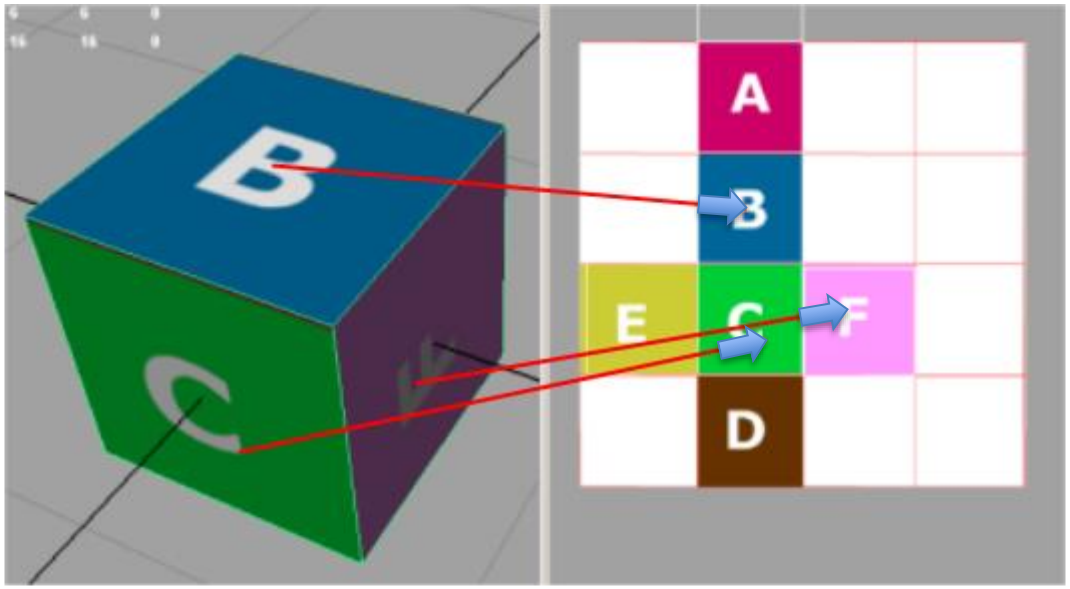


Texture

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

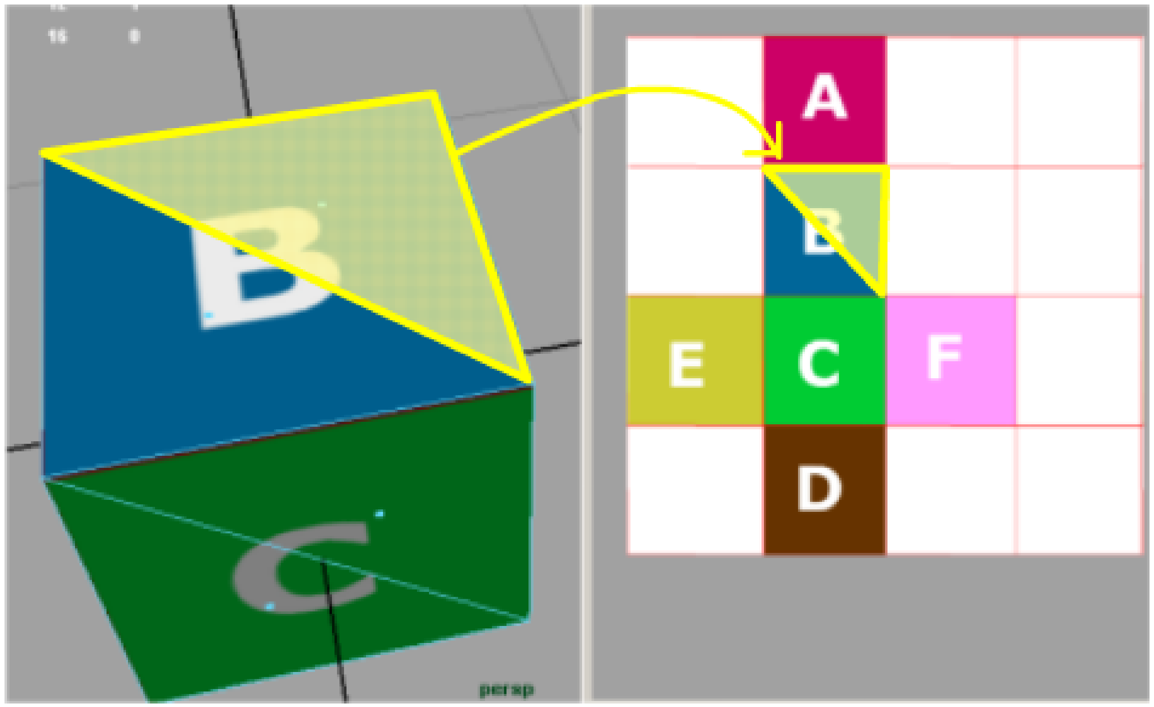
The mapping creates a correspondence between each point on the object, with a pixels of the texture (called *texel*).

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 meshes, the mapping procedure creates a correspondence between triangles on the surface, and triangles over the texture.

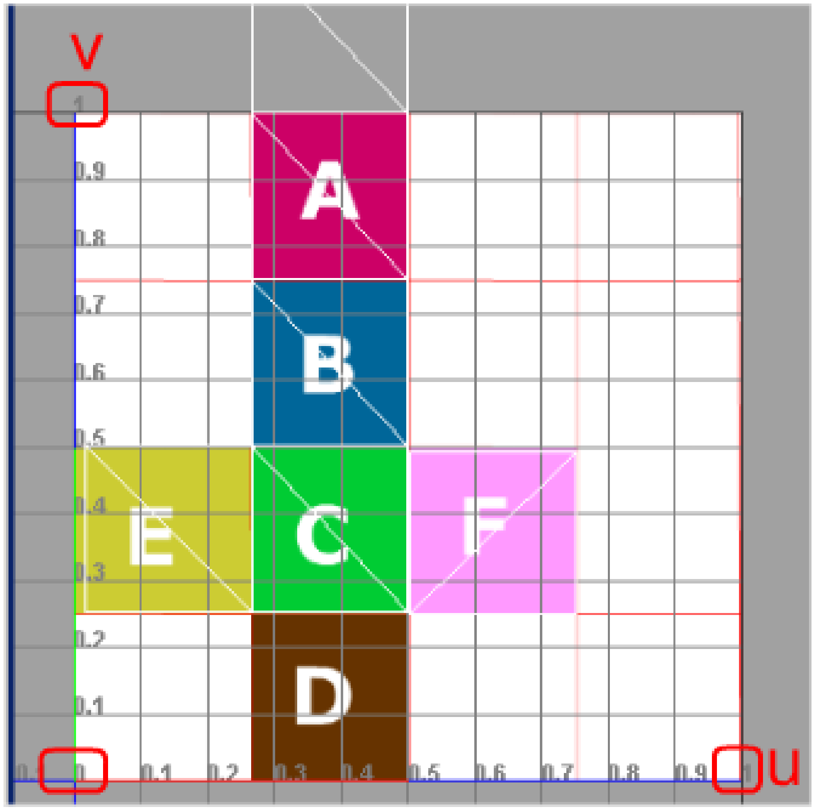


UV coordinates

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

This set of coordinates are then addressed as *UV, mapping or texture coordinates*.

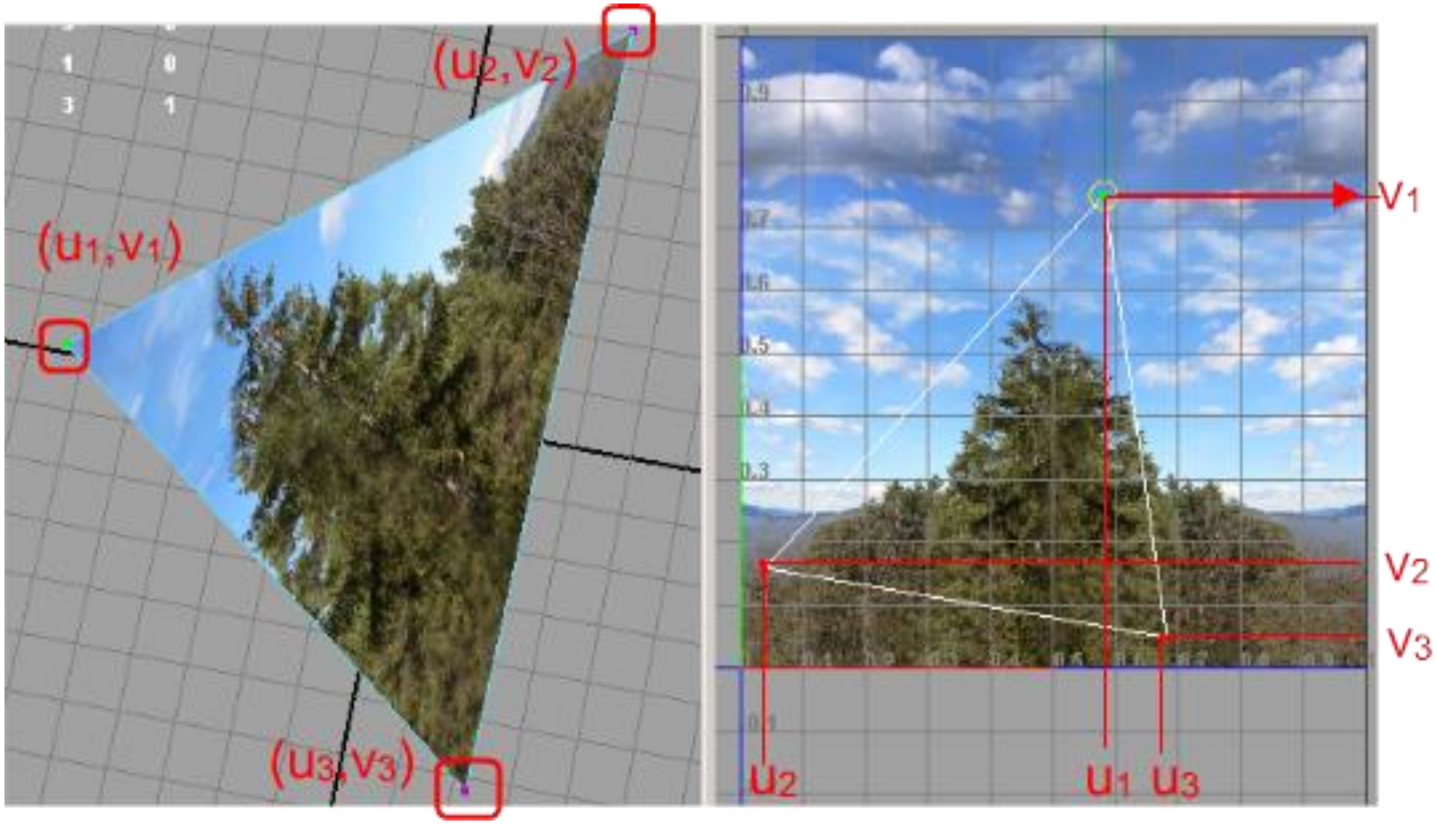
As a convention, the *u* and *v* values vary between *0* and *1* along the horizontal and vertical axes of the texture.



Note: sometimes an alternative axis naming convention, i.e. S and T. In other contexts, both notations are used with different meanings. In this course we will not enter in this detail, and simply use the UV notation.

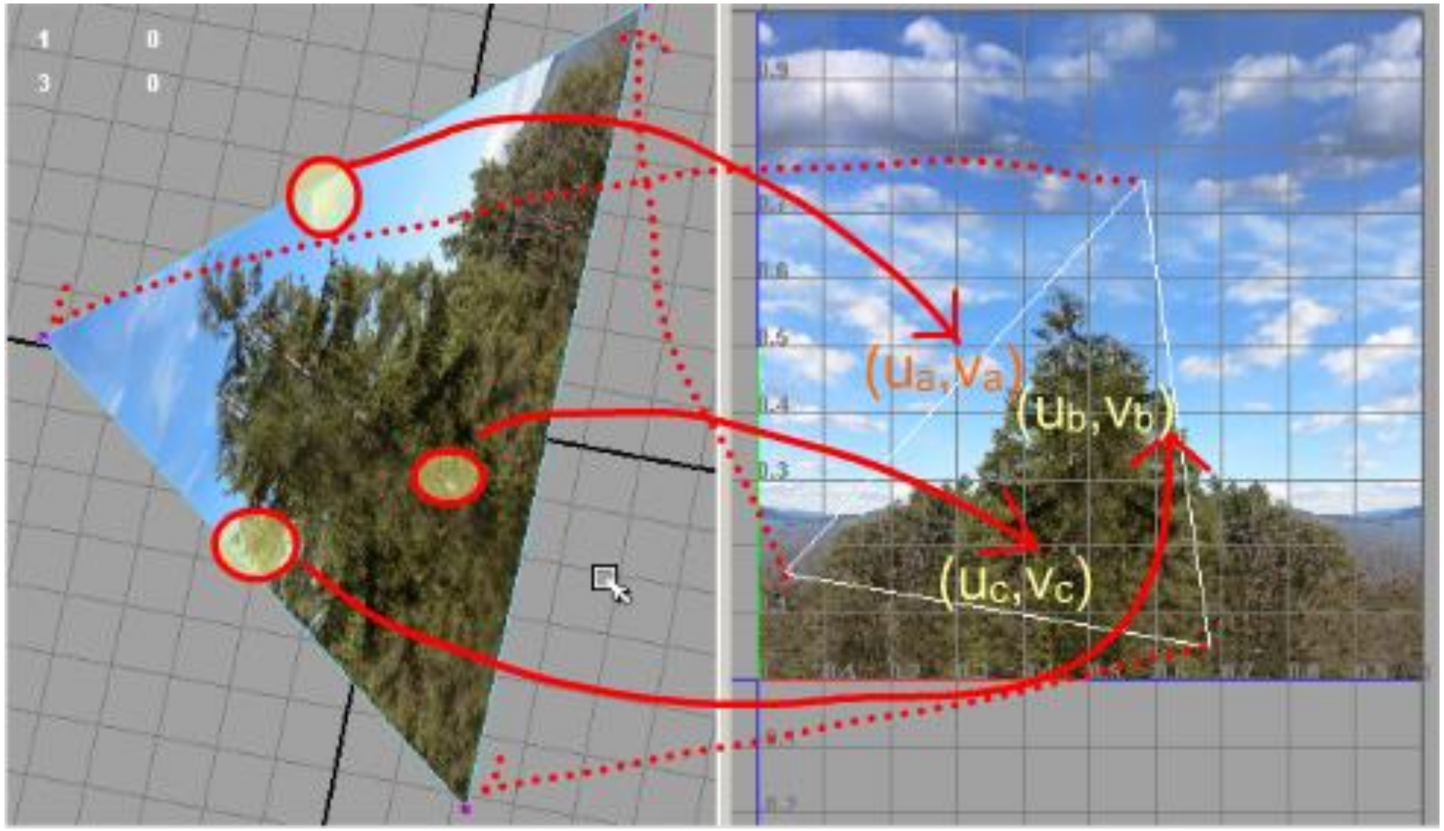
UV coordinates

UV coordinates are assigned *only to the vertices of the triangles*, adding two extra parameters to the position (and usually the normal vector direction).



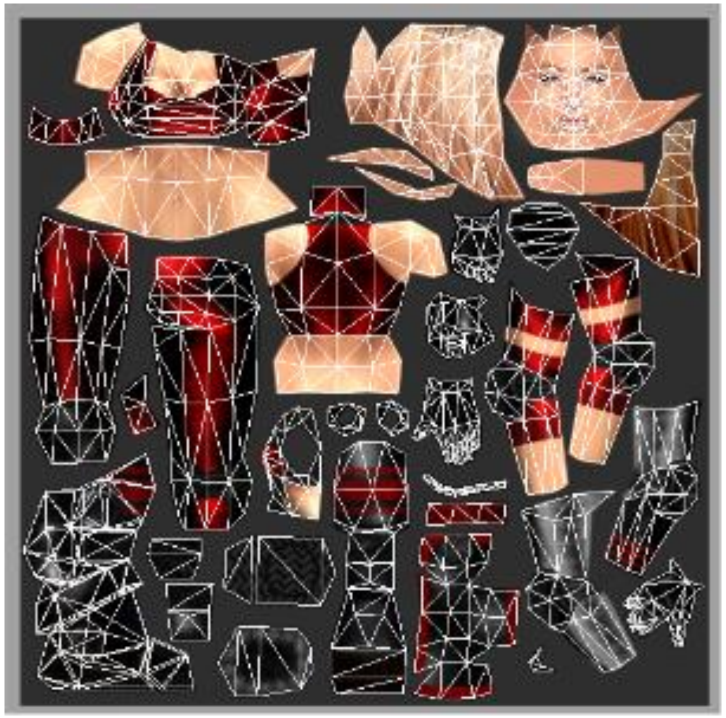
UV coordinates

For the internal points, the values of the UV coordinates are computed via interpolation, using the same technique seen for normal vector directions.



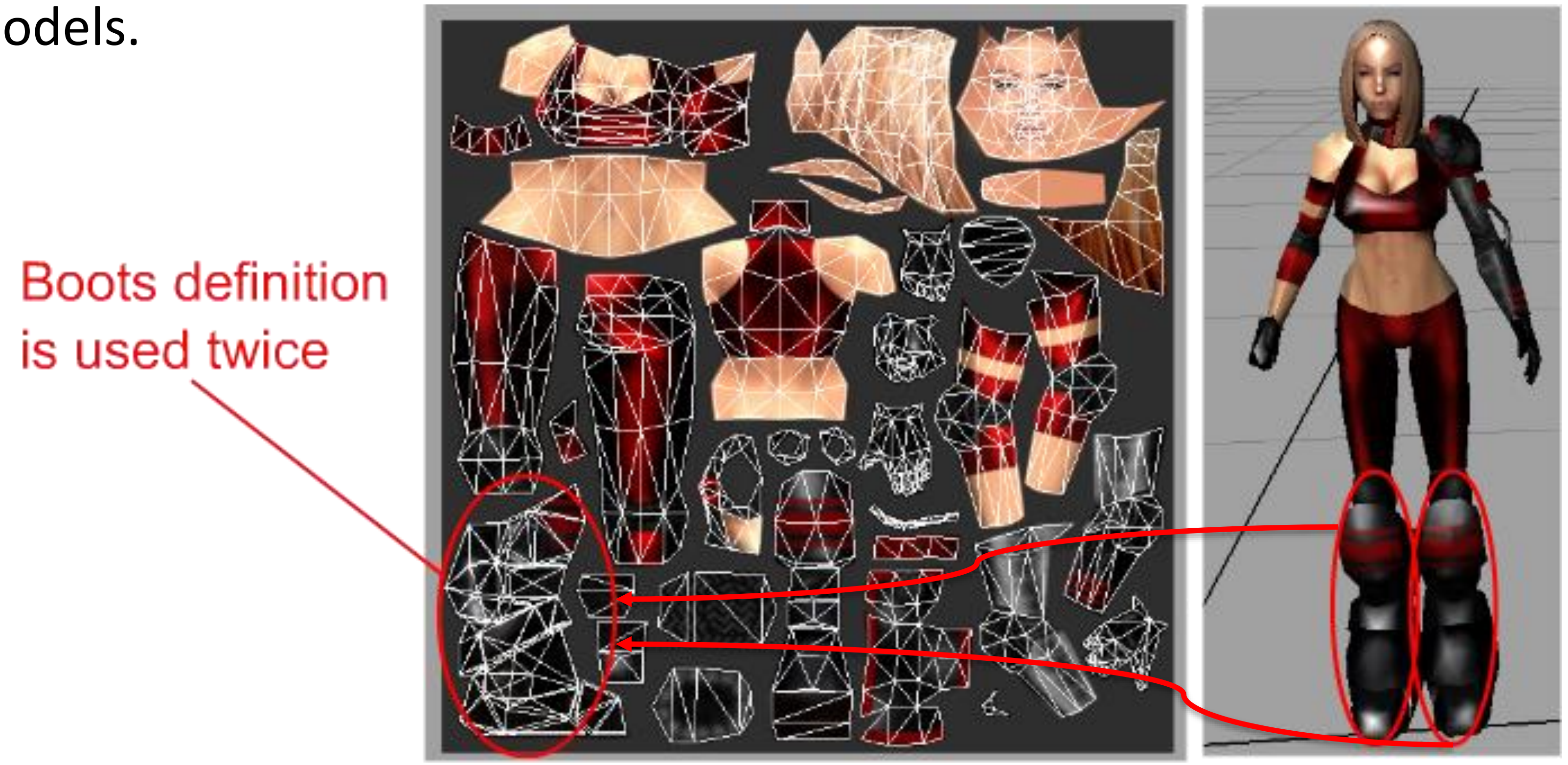
UV coordinates

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.



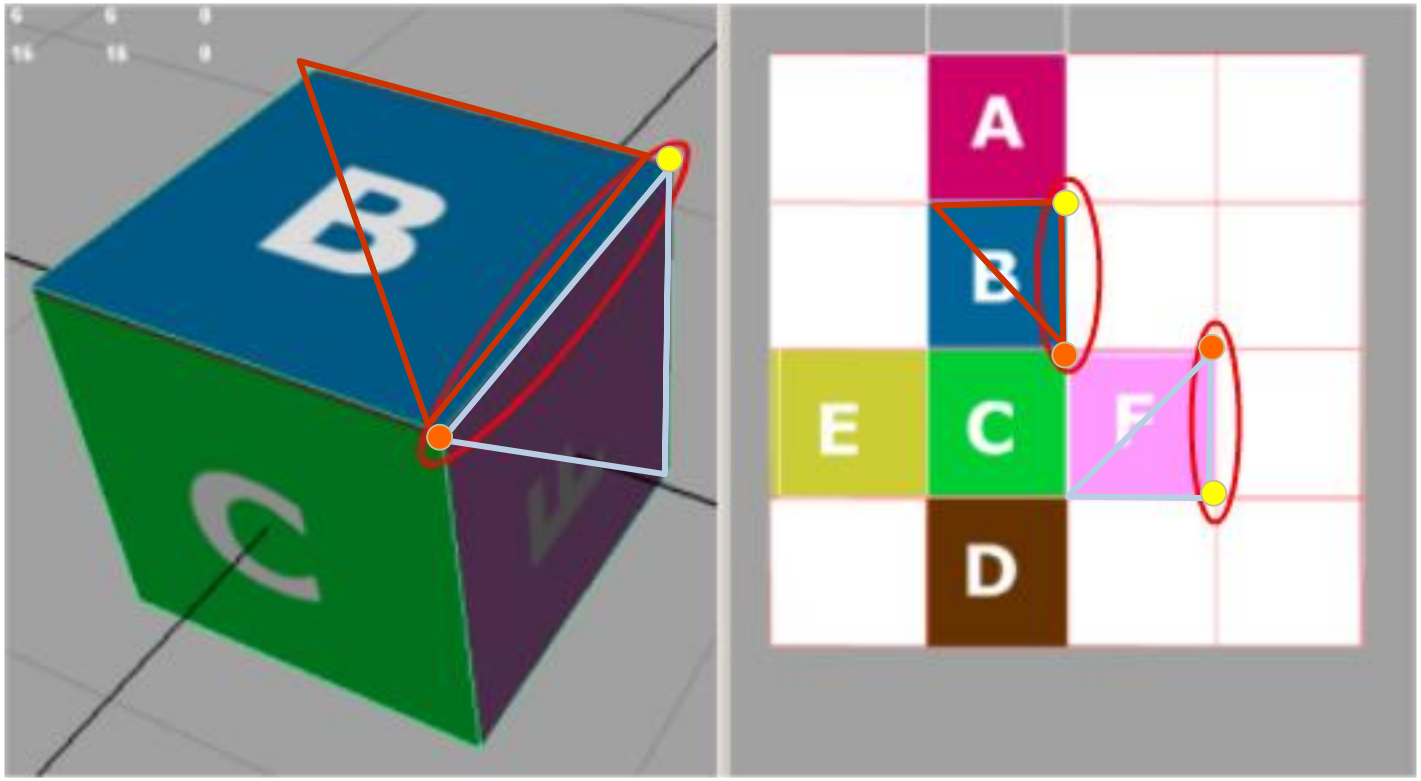
UV coordinates

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 3D artists can exploit this feature to improve the quality of their models.



UV coordinates

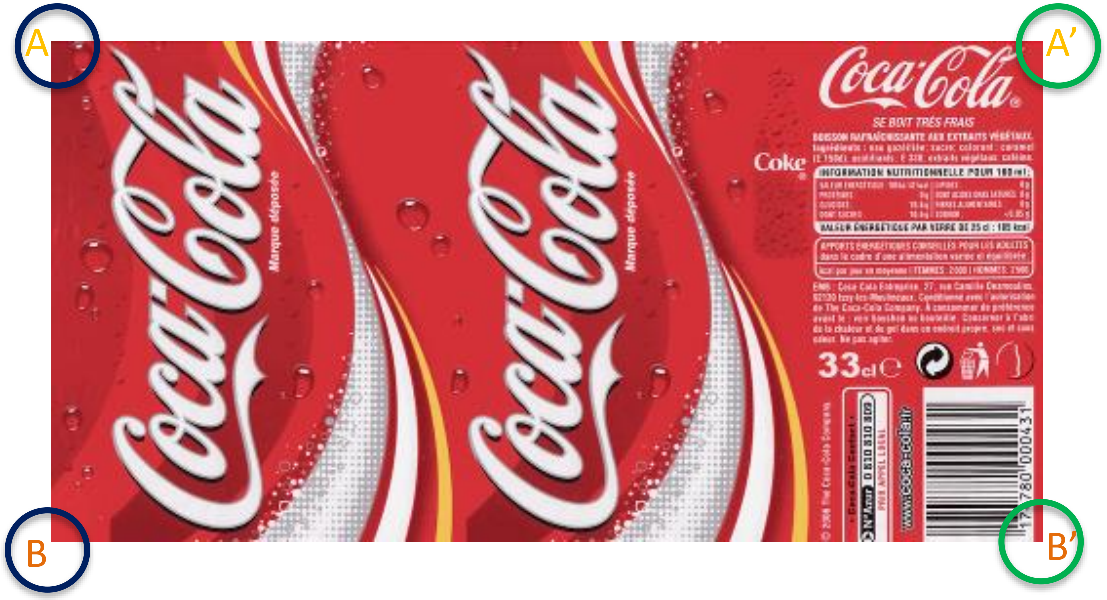
Note that, as seen for normal vector directions, two vertices belonging to two different triangles may have the same position in space, but different UV coordinates.

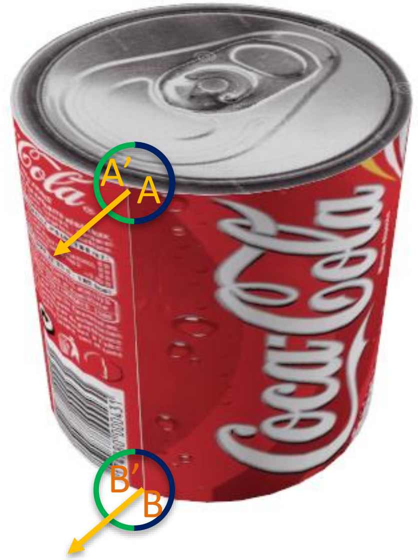


POLITECNICO MILANO 1863

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, they will have vertices characterized by the same position and the same normal vector direction, but with different UV coordinates.



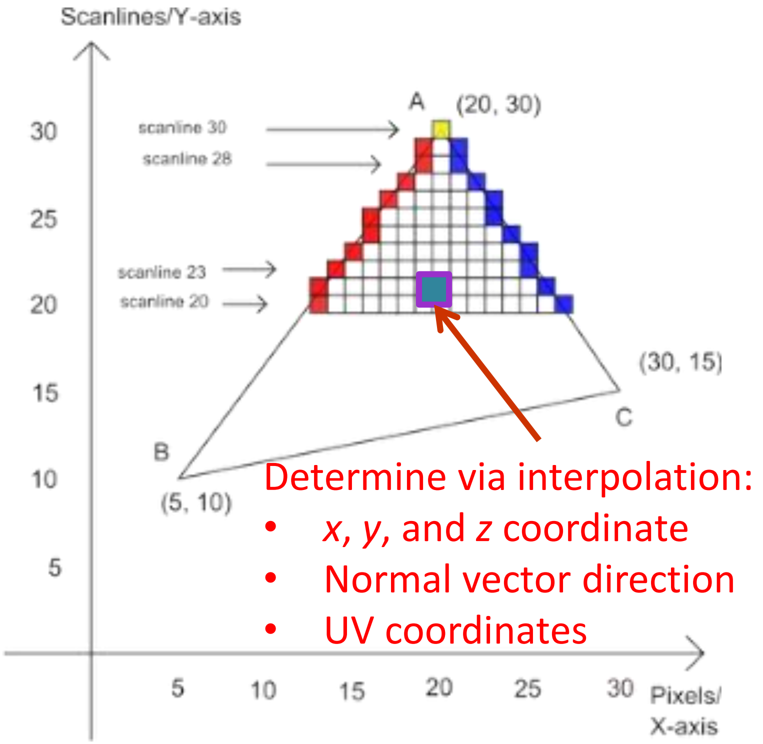


Rasterisation with textures

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.



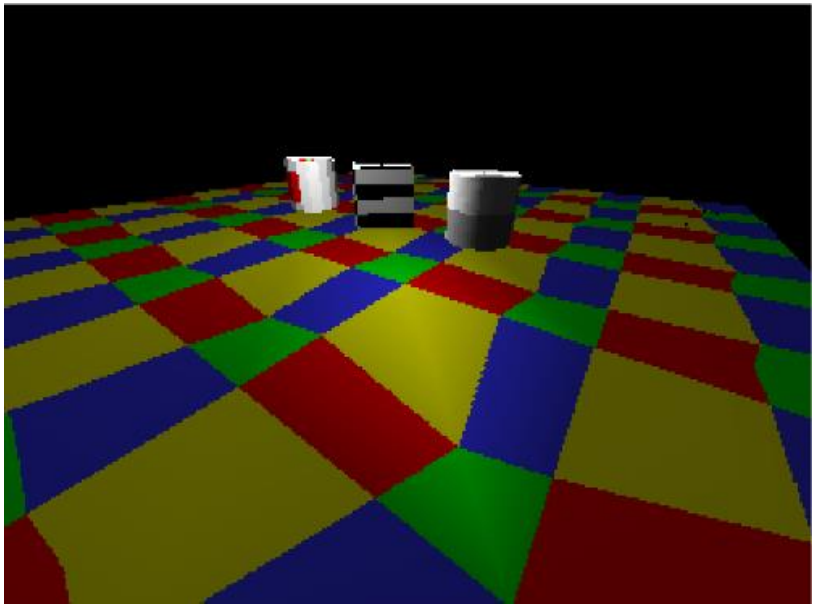
Determine via interpolation:

- *x*, *y*, and *z* coordinate
- Normal vector direction
- UV coordinates

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 the parameters associated to triangles when using perspective: straight lines appear to be wobbling.



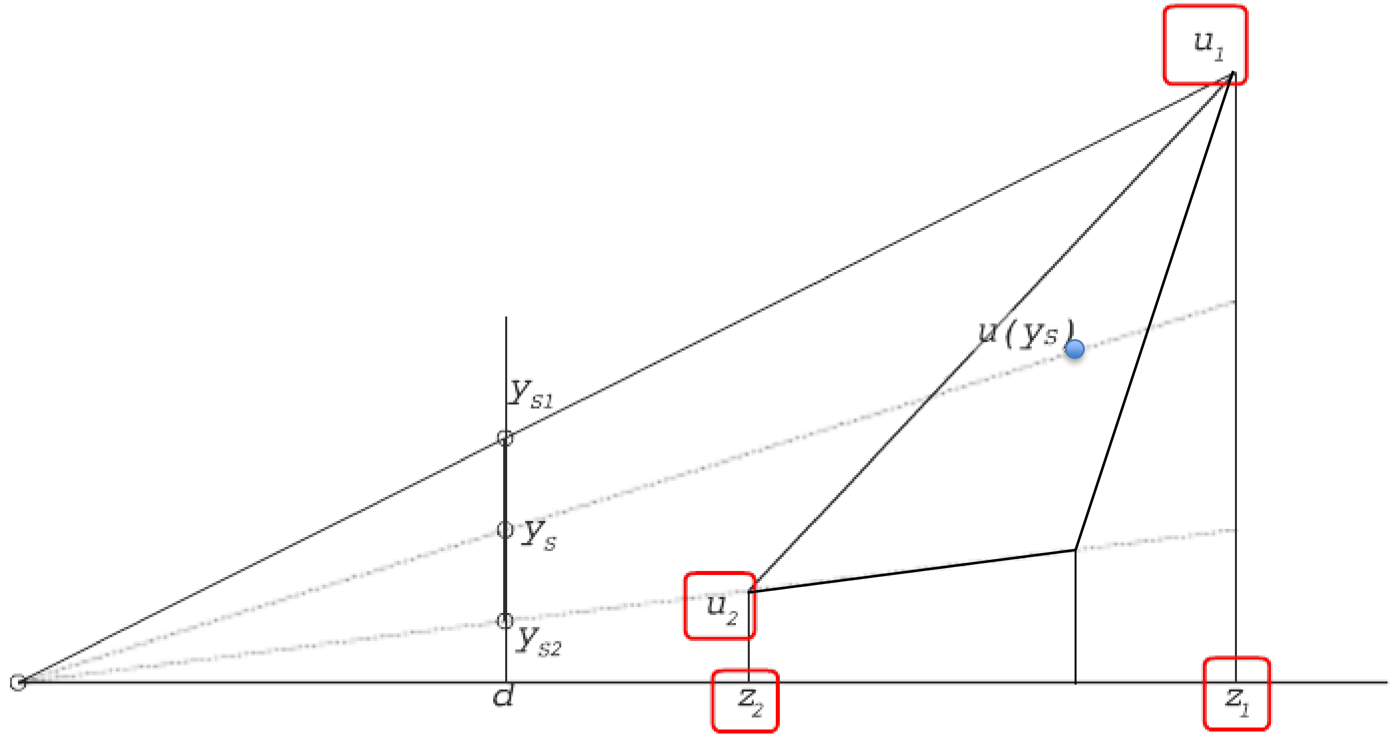
Perspective interpolation

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.

Perspective correction: derivation [just for completeness]

Let us consider a parameter *u*, which can be any value associated to a texture coordinate, a position, a color or a normal vector direction, and which assumes values u_1 at distance z_1, u_2 at z_2, and u_3 at z_3.



Perspective correction: derivation [just for completeness]

As introduced, an internal point *(x_S,y_S,z_S)* of a triangle can be considered as a linear combination its three vertices *(x_1,y_1,z_1), (x_2,y_2 ,z_2), (x_3,y_3 ,z_3)* with coefficients that sums up to one.

$$(x_S, y_S, z_S) = (x_1, y_1, z_1) \cdot \alpha_1 + (x_2, y_2, z_2) \cdot \alpha_2 + (x_3, y_3, z_3) \cdot \alpha_3$$

with:

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

It can be shown that the value u_s at *(x_S,y_S,z_S)* with *perspective correct interpolation* can 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 [just for completeness]

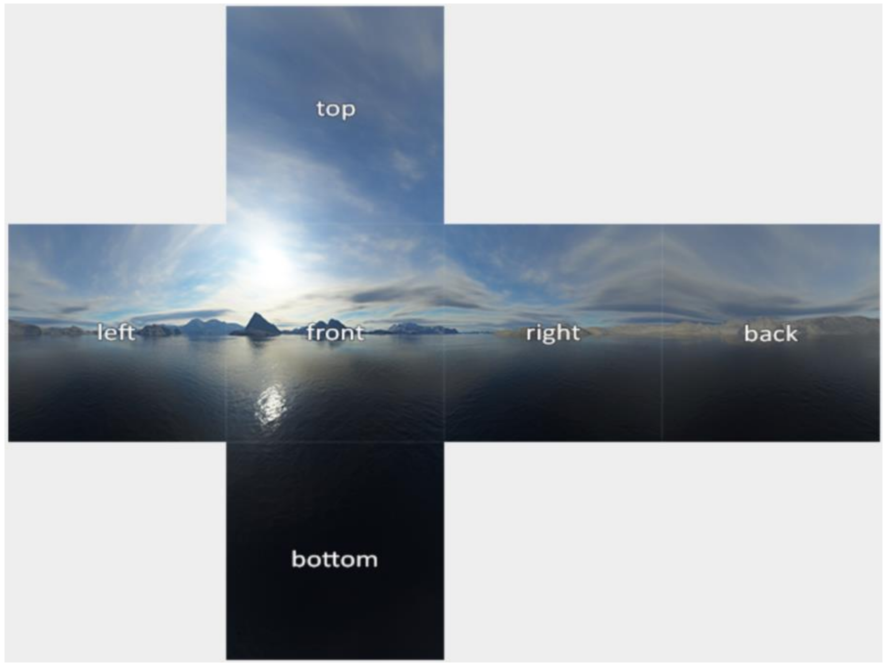
Perspective correct interpolation should be performed for all parameters considered *after the perspective projection*.

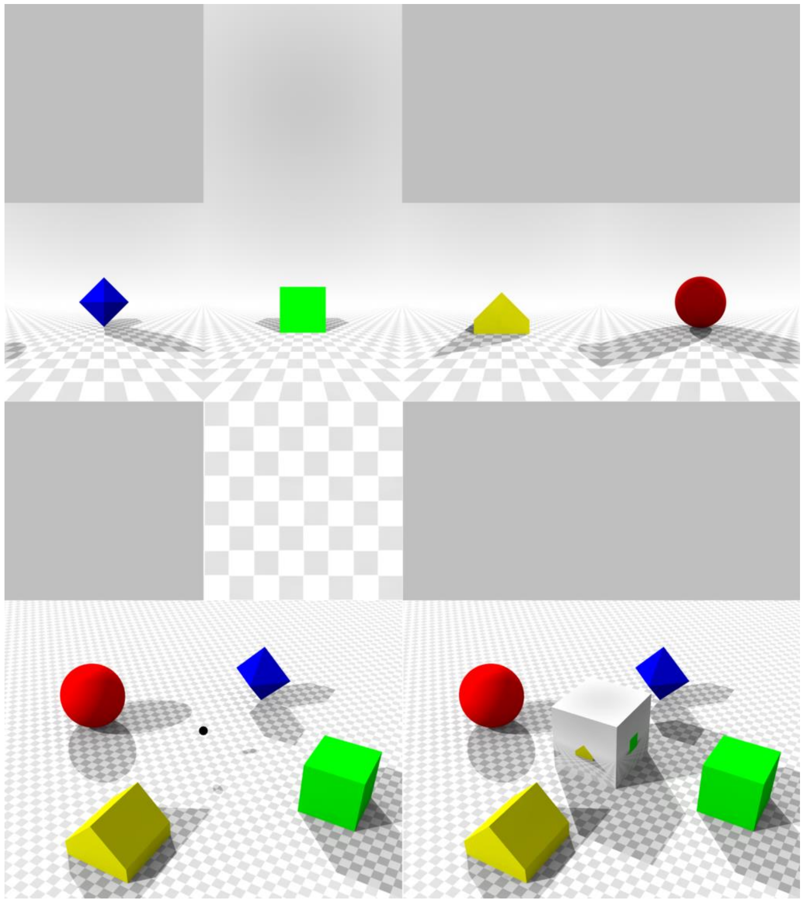
As seen, perspective correct interpolation requires the distance of the point from the center of projection along the negative z axis: in clipping coordinates (i.e. before the normalization step), this corresponds to the fourth component of the homogenous coordinate vector changed of sign.

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

Cube map textures

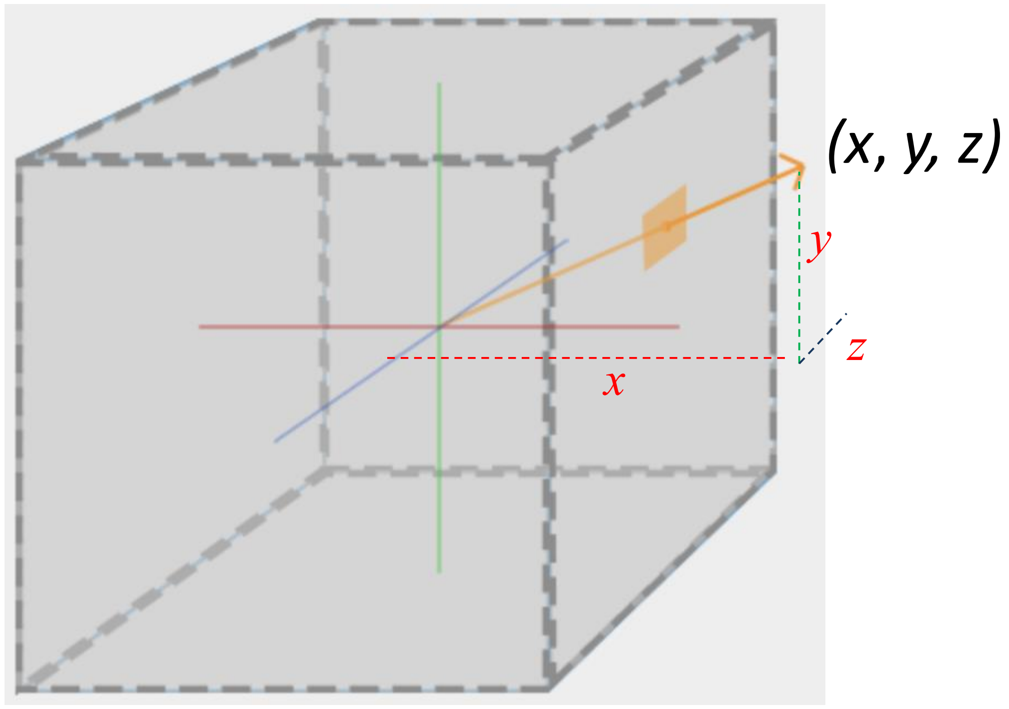
Cube-map textures are collections of six images, that correspond to the six sides of a cube.





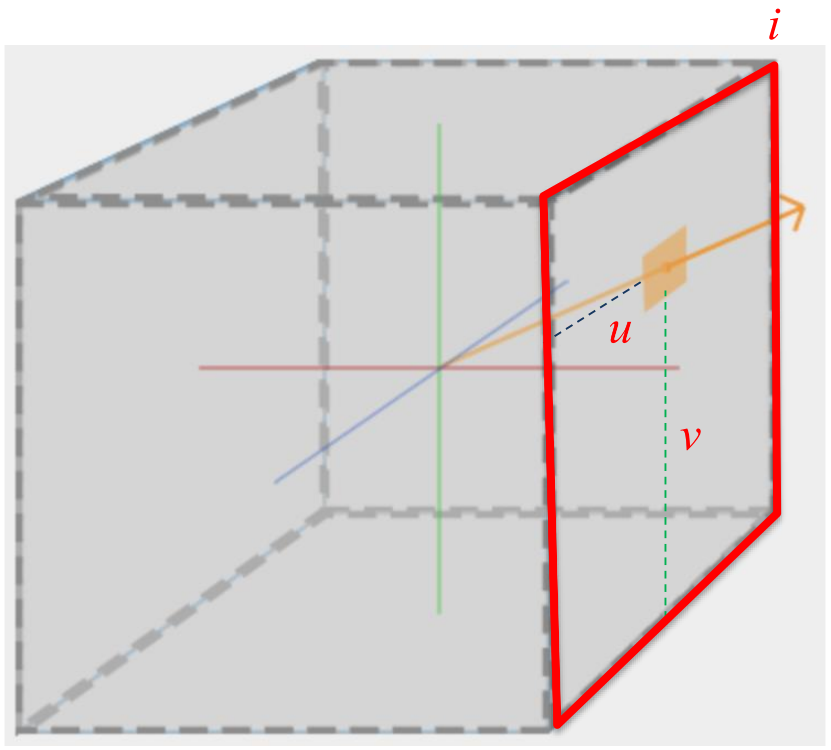
Cube map textures

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



Cube map textures

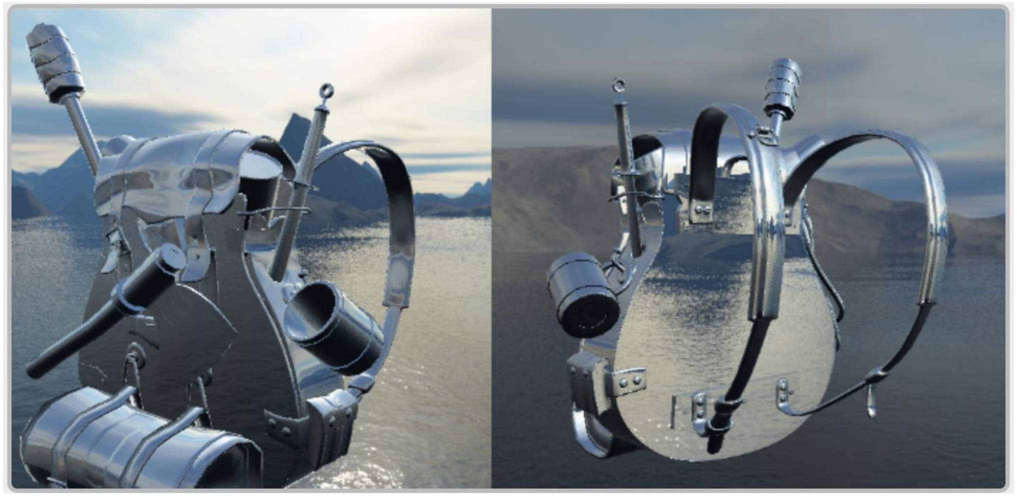
The normal vector components determine both the side of the cube (the base image), and the corresponding texel on such image.



Cube map textures

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

They are also used to store image based ambient lighting parameters.



UV intervals

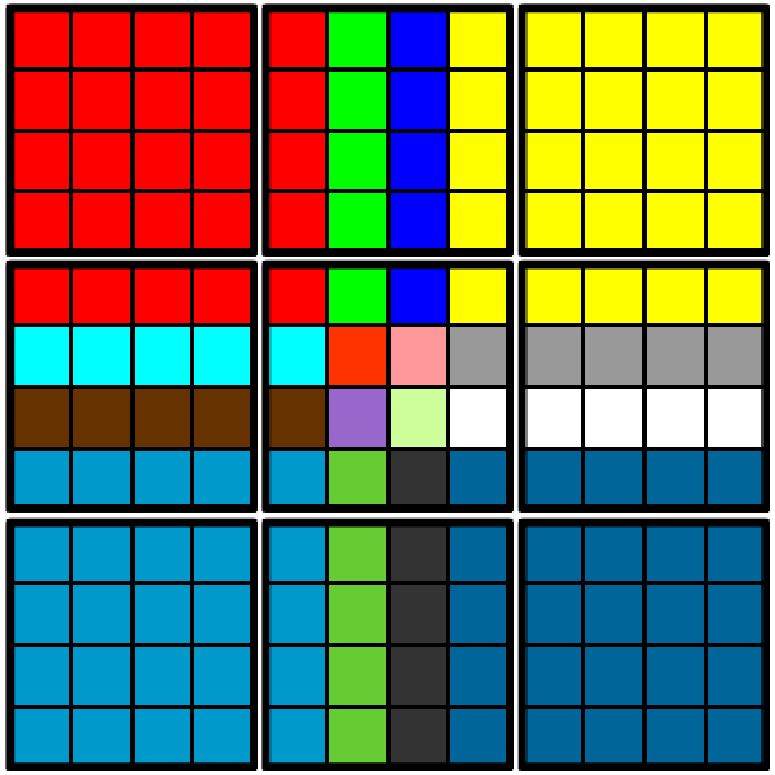
In many interesting cases, the *u* or *v* values fall outside the *[0, 1*] range.

This can be interpreted in several ways, which can have a different impact on the final value fetch from the texture. The four most popular ways of handling values that are negative or greater than one are:

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

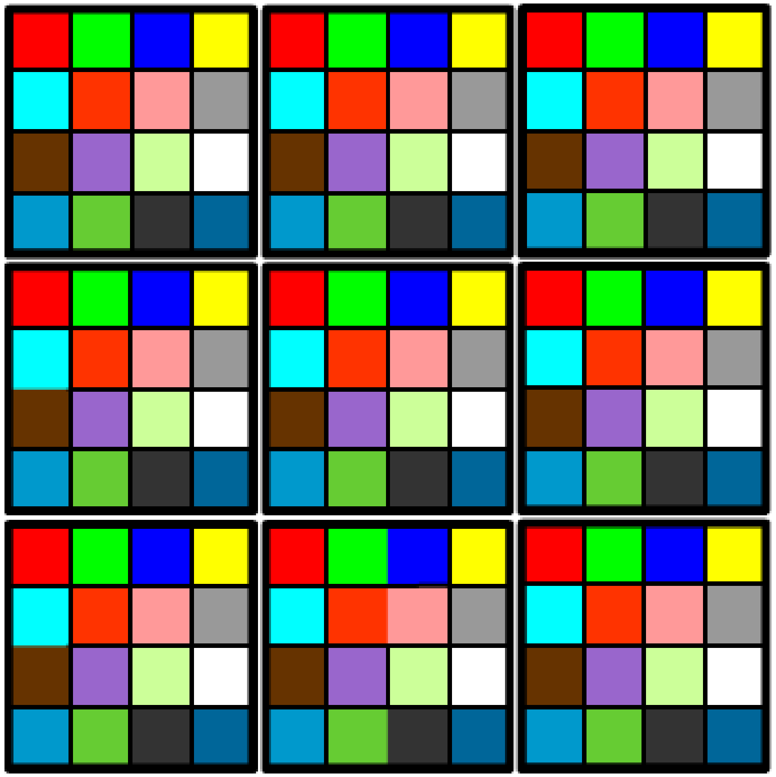
Clamp

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



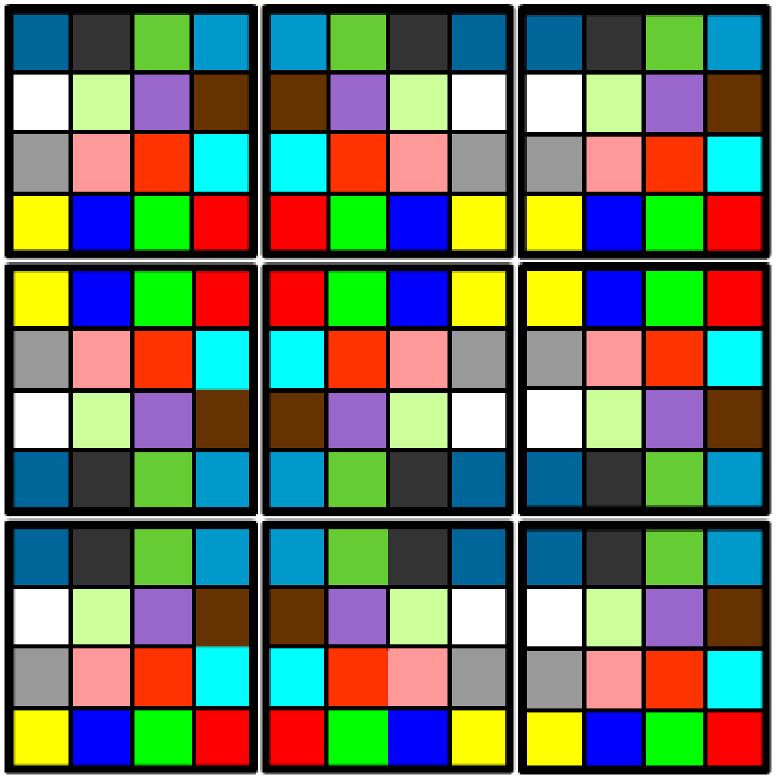
Repeat

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



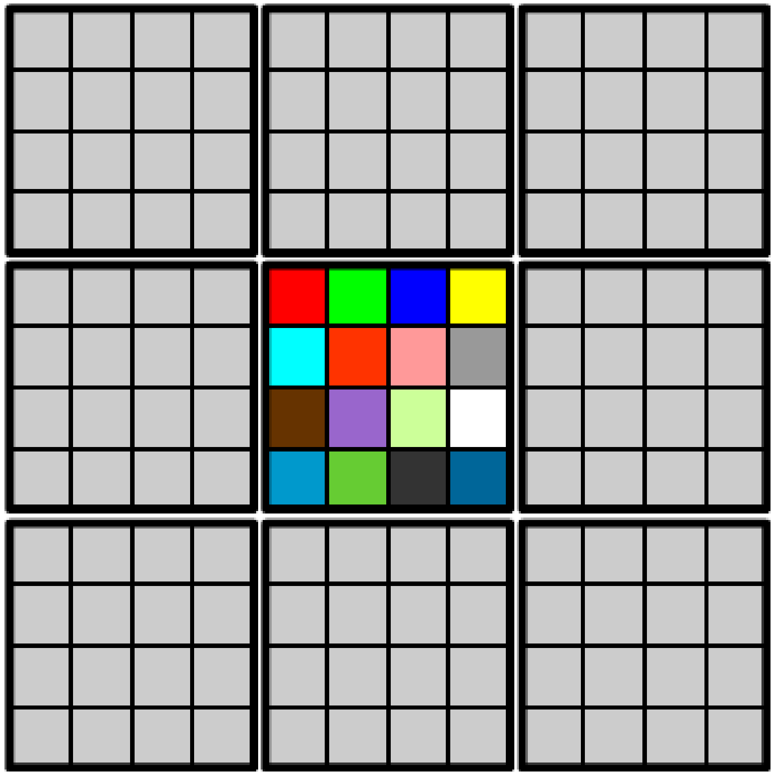
Mirror

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



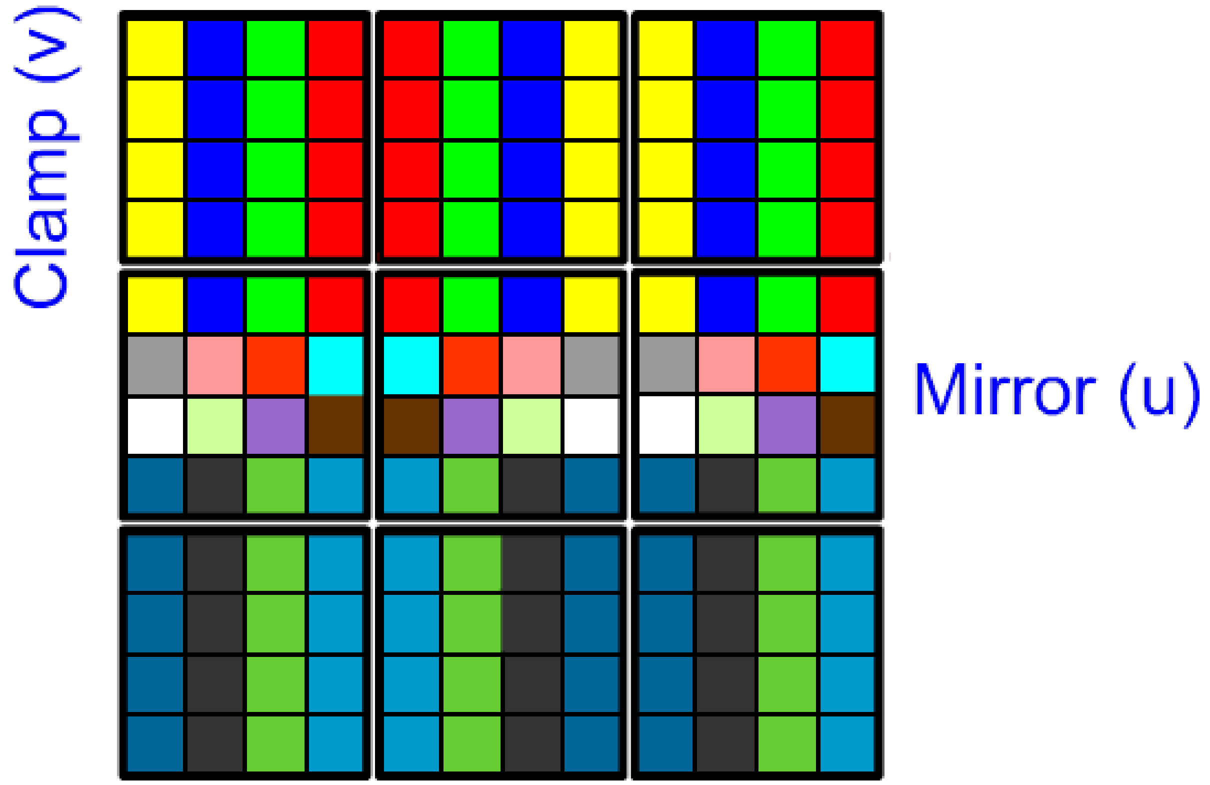
Constant

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* directions, several effects can be obtained.



Texture filtering

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

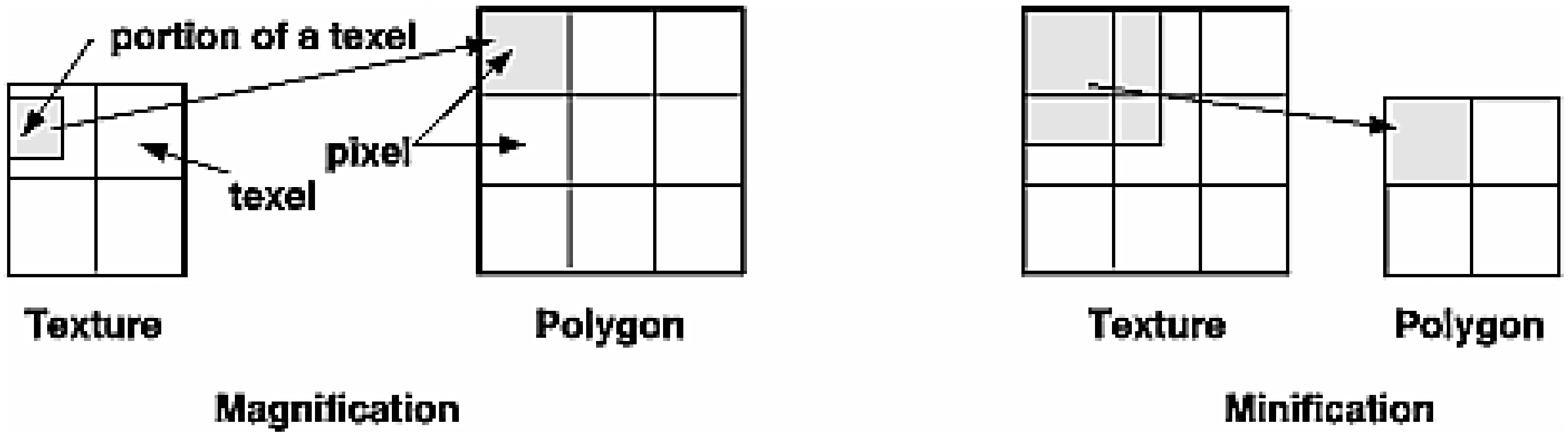
Moreover, the shape of a pixel on the screen might correspond to a complex polygon composed by several texels on the texture.

Such problems are solved using a set of techniques known as *filtering*.

Texture filtering

When a texel is larger than the corresponding pixel on screen, 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 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.* The 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 one corner of the image. In Vulkan, this is the top left corner.

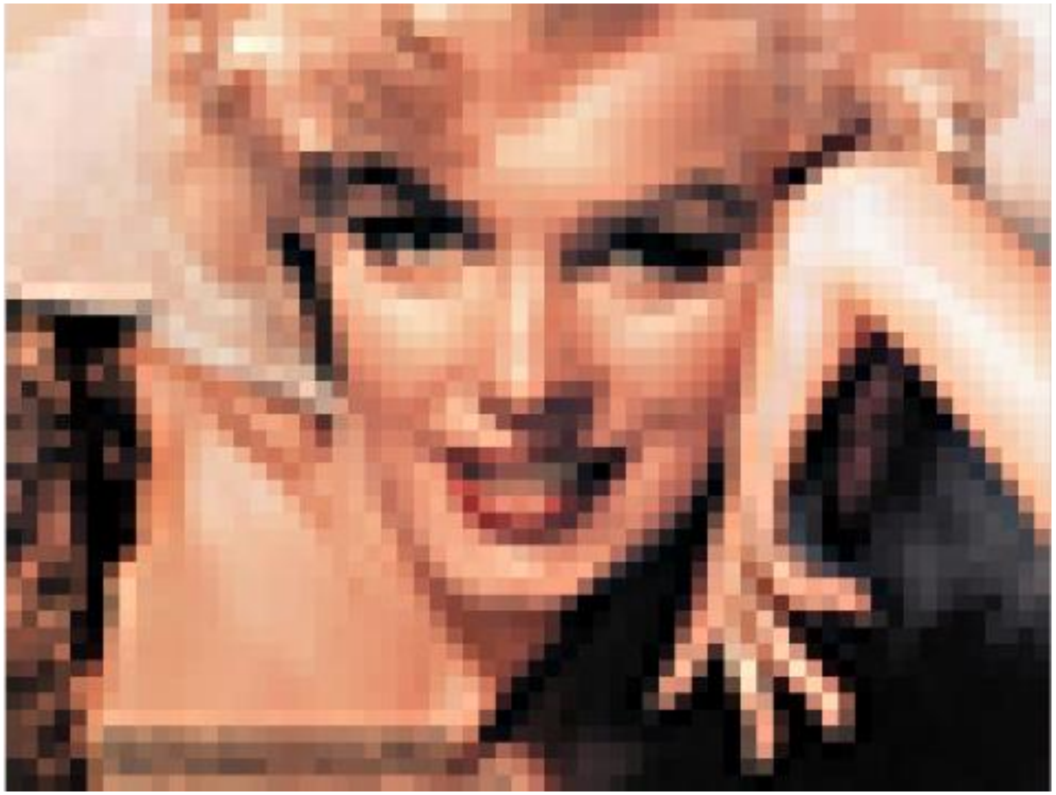
With the *nearest pixel* filter, the lookup procedure first transforms the UV coordinates with respect to the texture size, then returns the texel *p[i][j]* that considers only the integer part of the scaled coordinates.

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

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

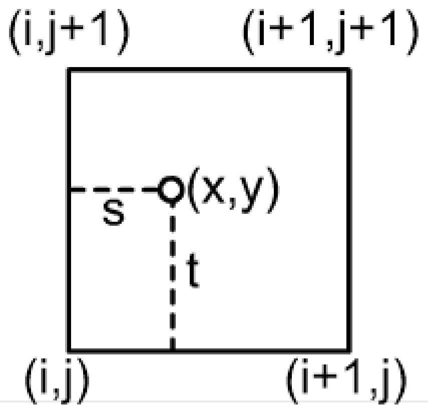
Nearest pixel

This technique 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.

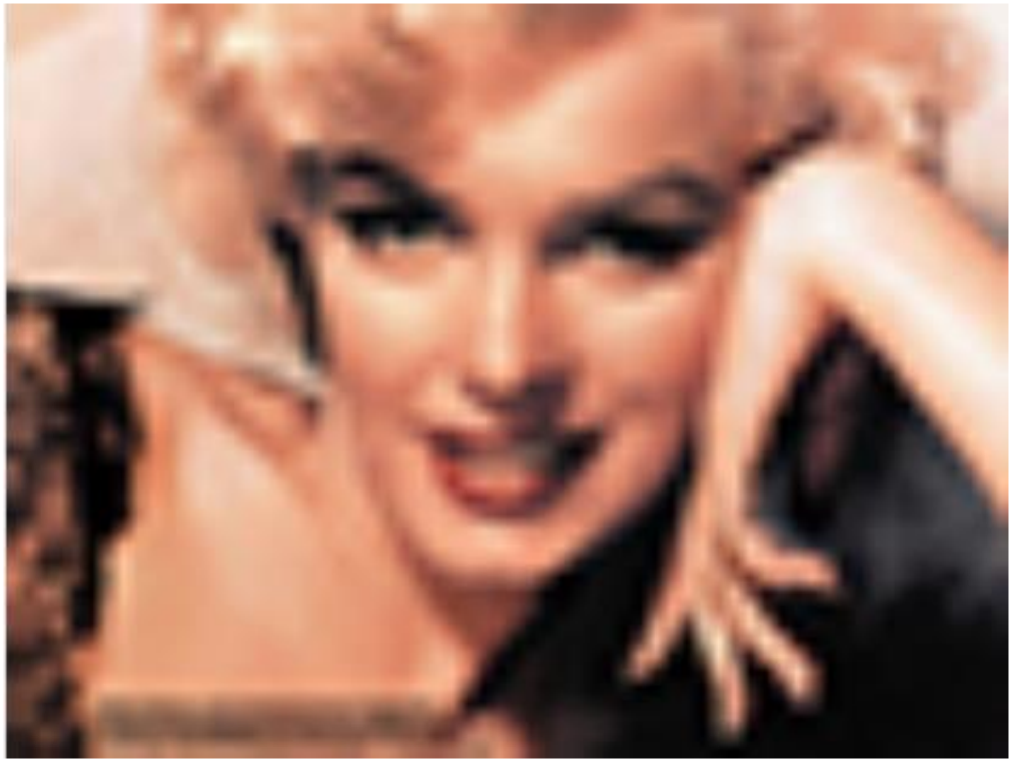


Please note that this figure refers to OpenGL. Vulkan has the vertical indices oriented in the opposite direction.

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

(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). This interpolation is even more complex in cube maps for pixels along the borders or at the corners.

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

Minification filters

The most common minification filters are:

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

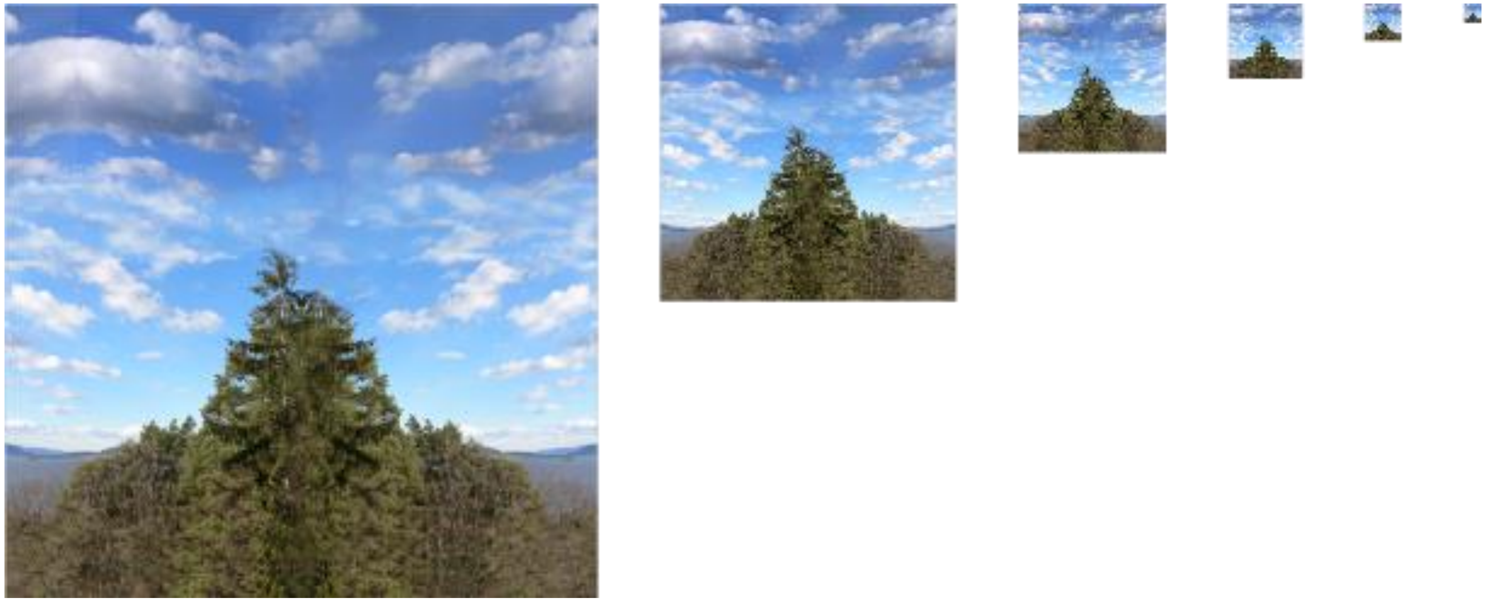
Nearest pixel and bilinear interpolation perform exactly in the same way as for the magnification: however they both give very poor results if the reduction is large.

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*).

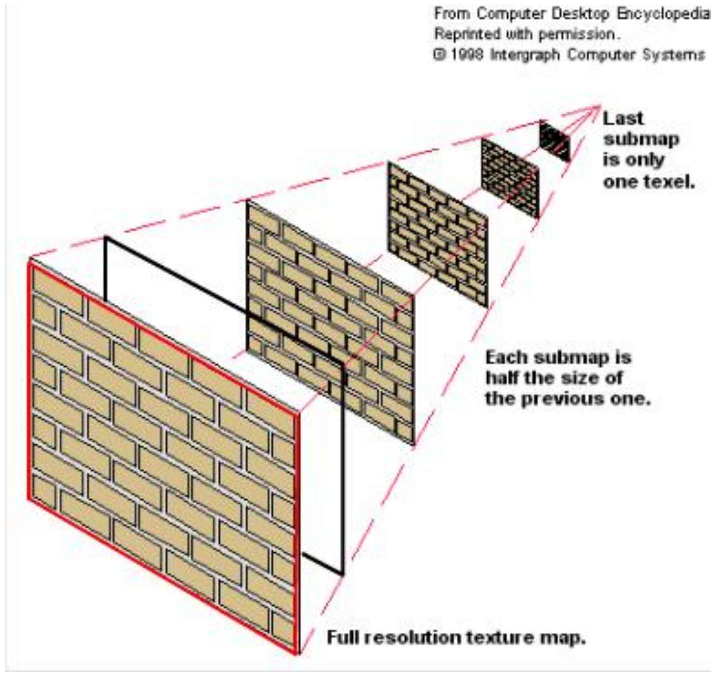
The technique pre-computes a set of scaled versions of the texture (called *levels*), each one with halved size.



Mip-mapping

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

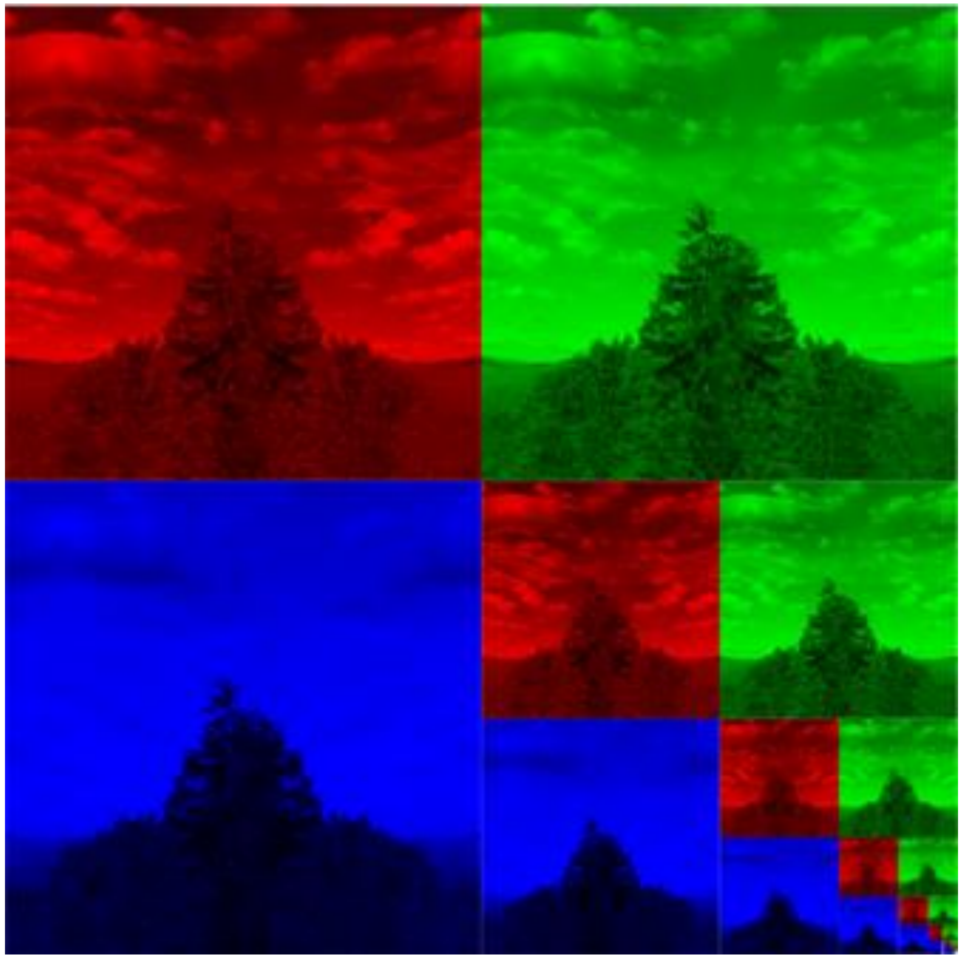
Depending on the ratio between the texture and the pixels, the algorithm selects the closest image in the mip-map.


From Computer Desktop Encyclopedia
Reprinted with permission.
© 1998 Intergraph Computer Systems
Last submap is only one texel.
Each submap is half the size of the previous one.
Full resolution texture map.

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.

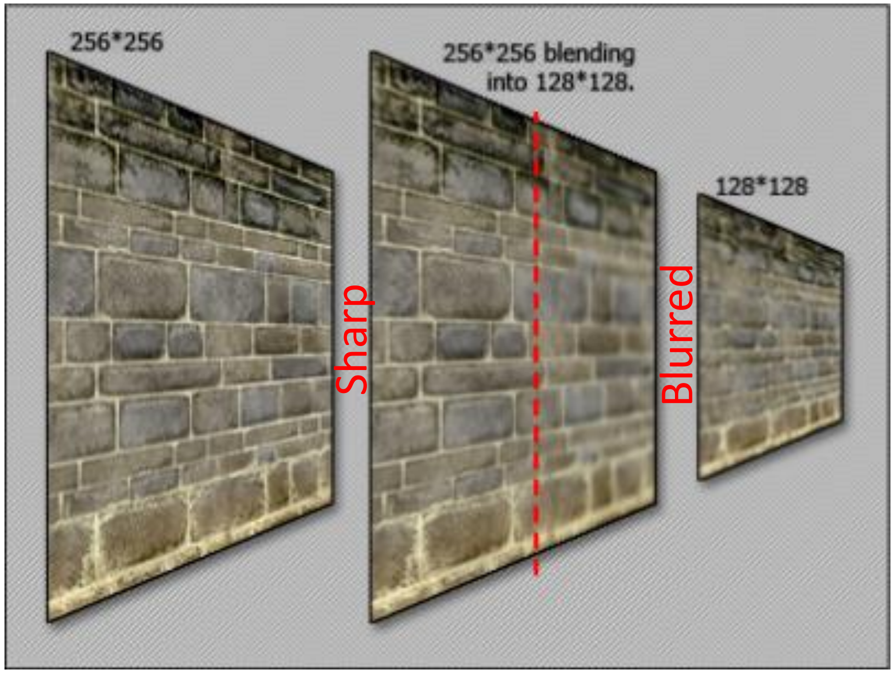
In Vulkan, this process must be manually programmed by the developer.



Trilinear filtering

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

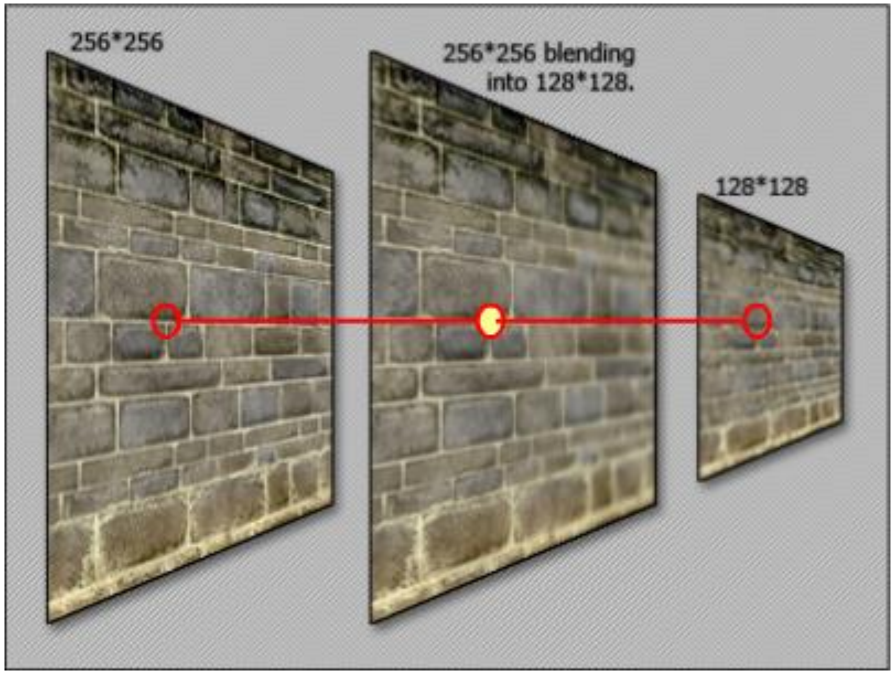
This might cause visible artifacts due to the abrupt change.



Trilinear filtering

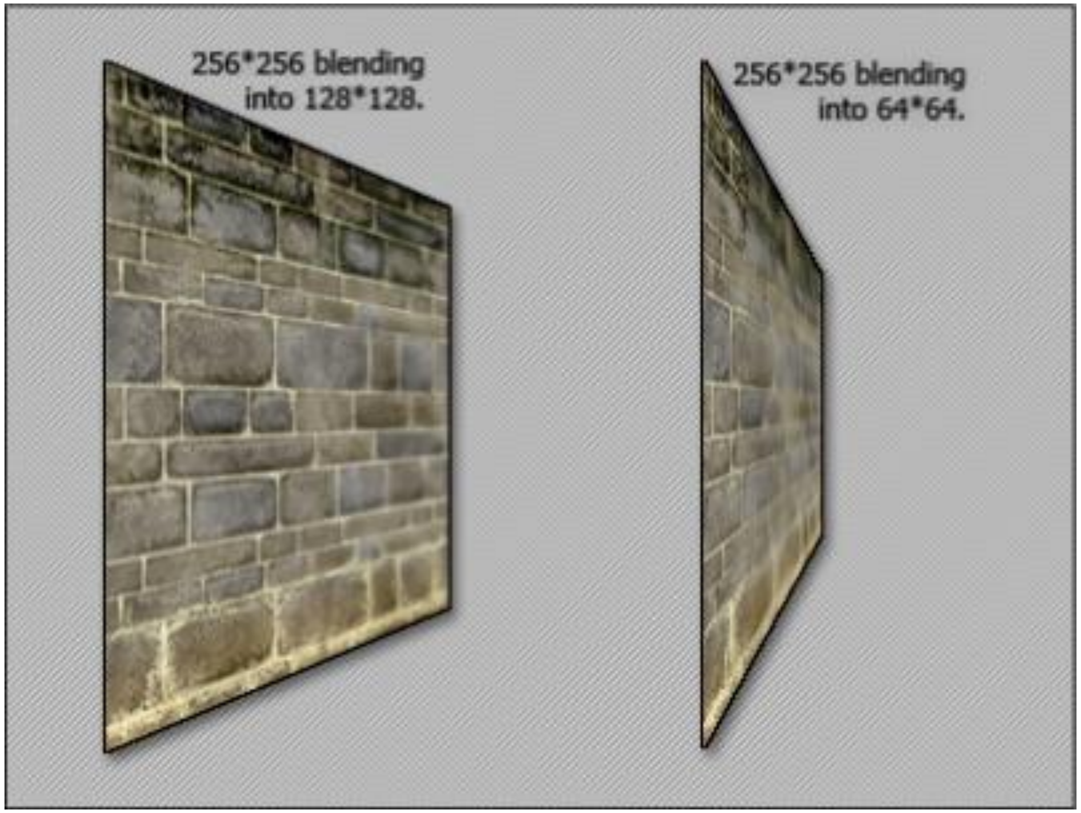
(Tri)linear filtering computes the required pixel two times, using the two closest levels of the Mip-map.

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



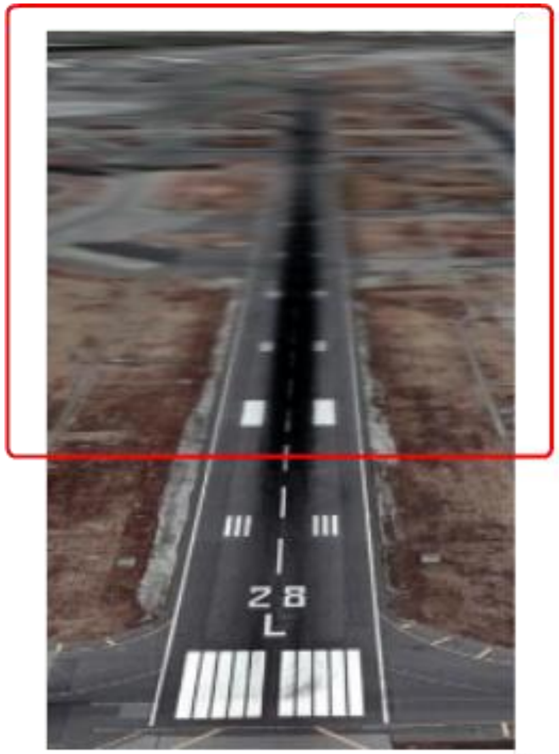
Trilinear filtering

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



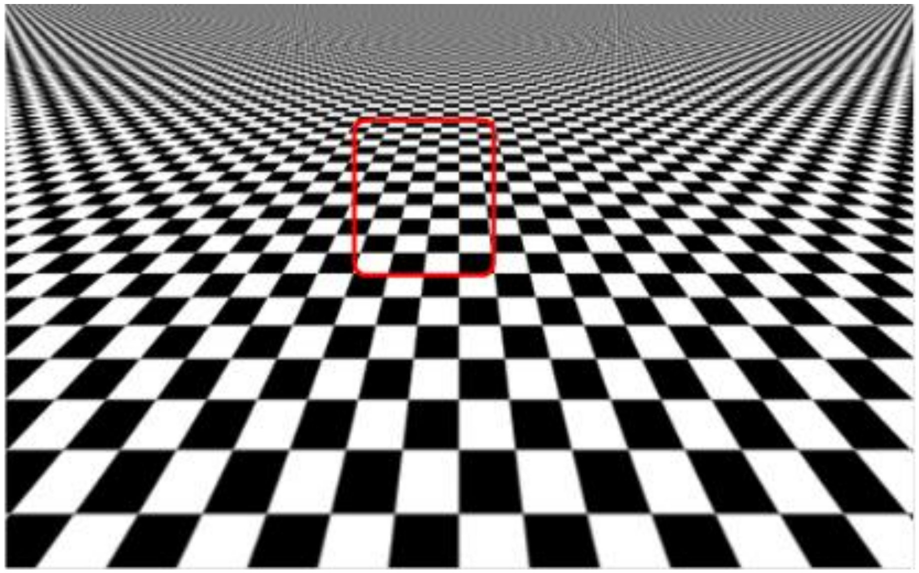
Rip-mapping

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



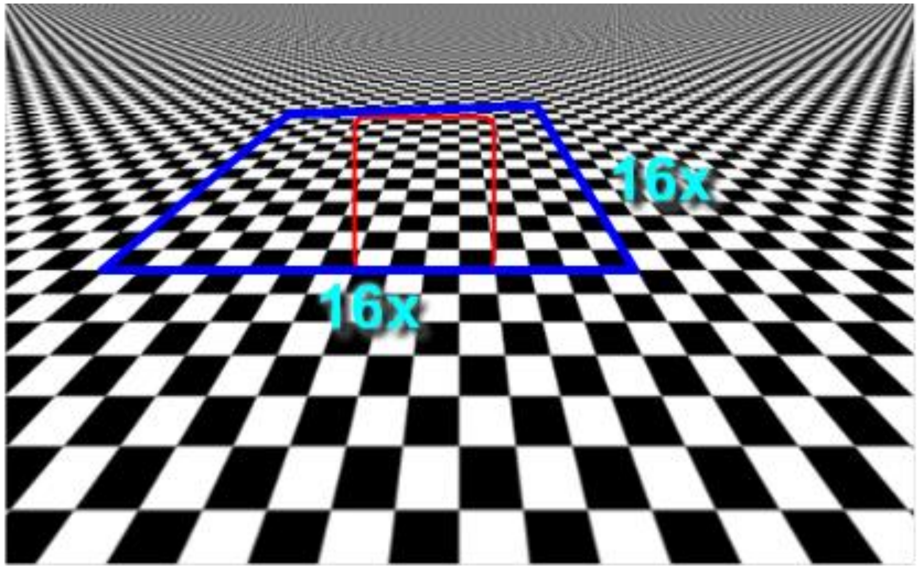
Rip-mapping

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



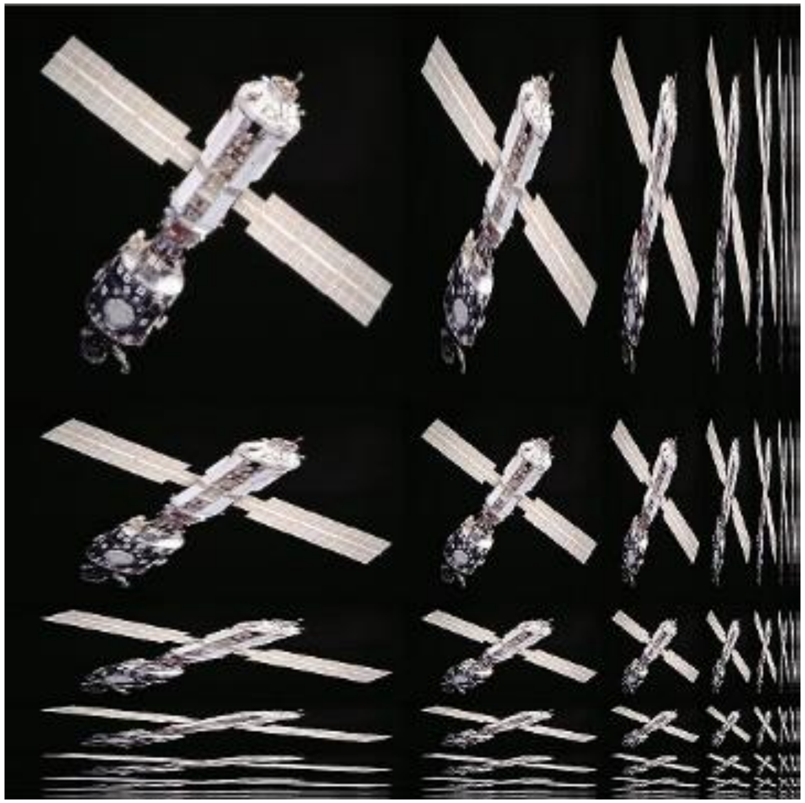
Rip-mapping

The texels of the Mip-map can be effectively used only for one direction of the rectangle: the other is definitely under-sampled.



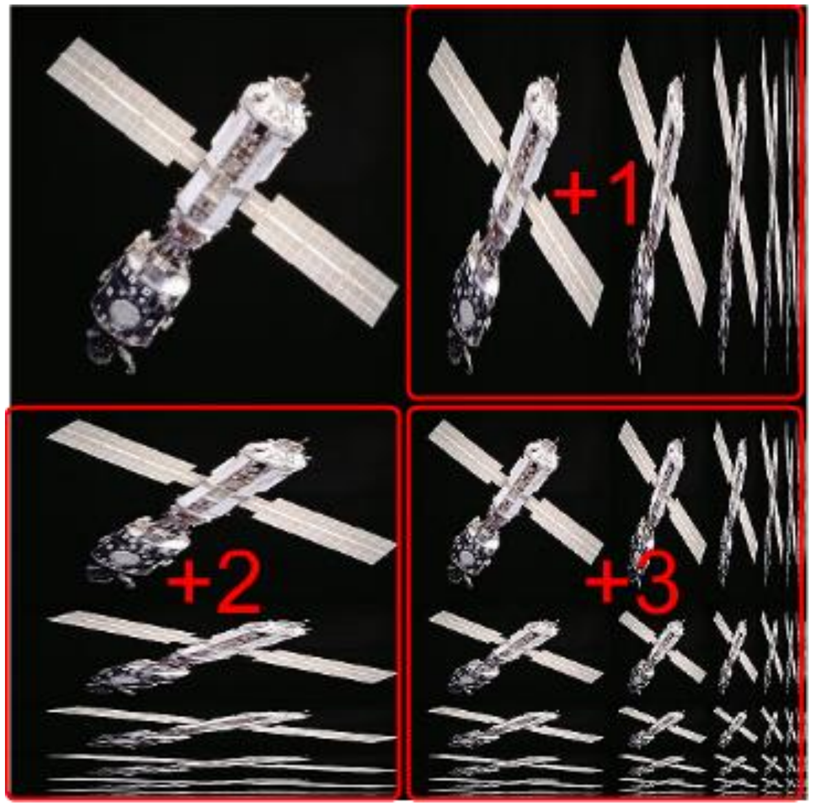
Rip-mapping

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



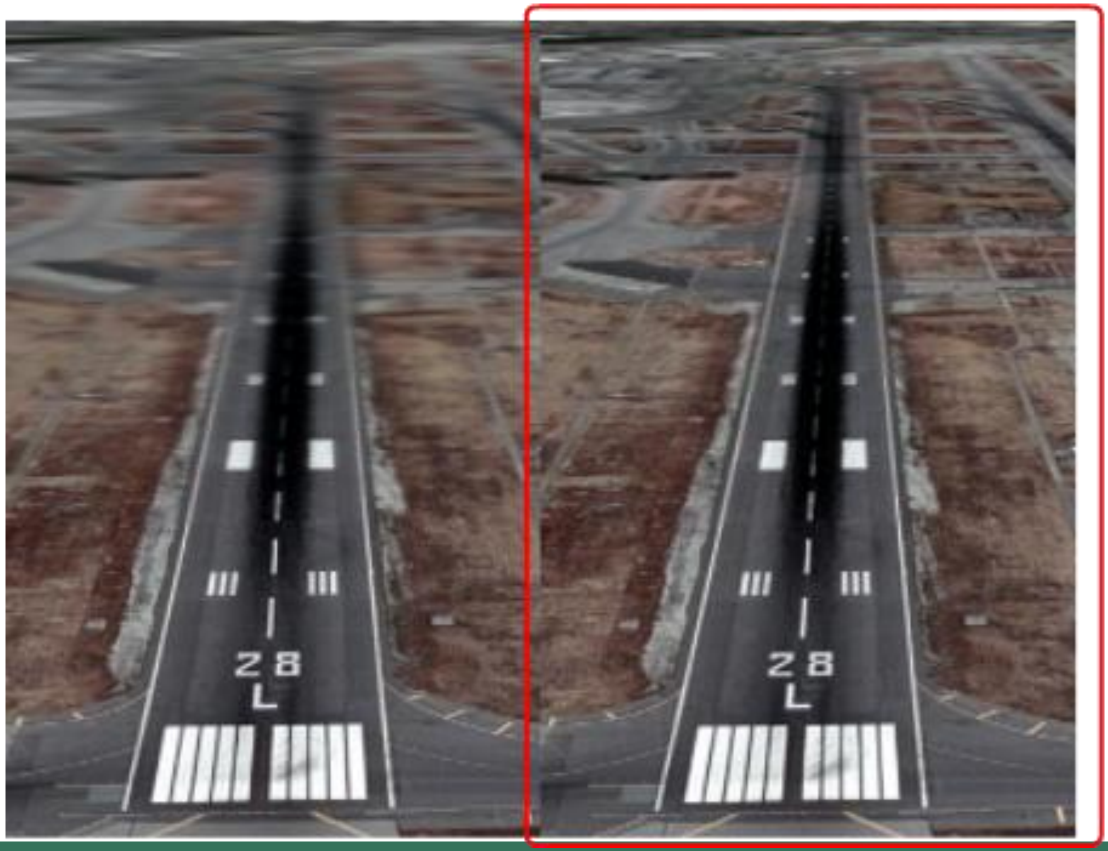
Rip-mapping

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



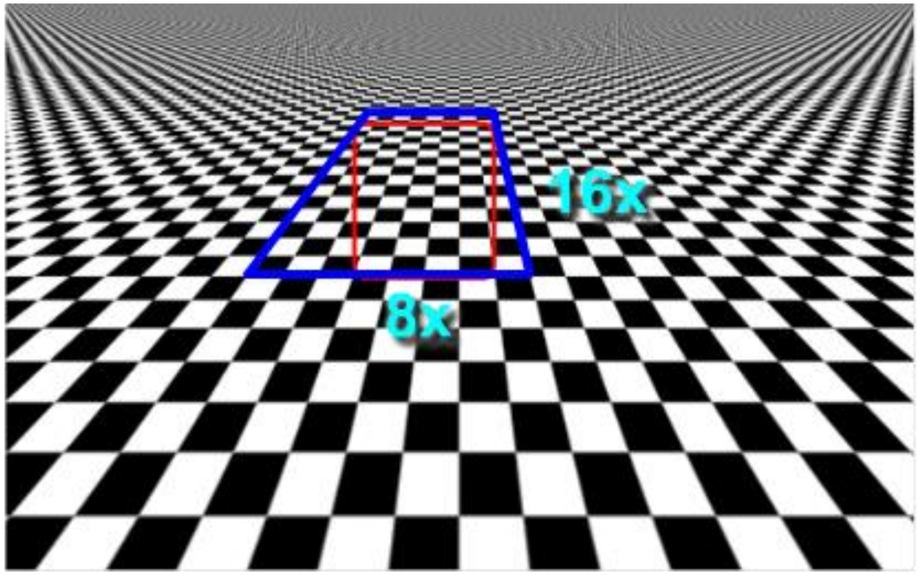
Rip-mapping

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



Rip-mapping

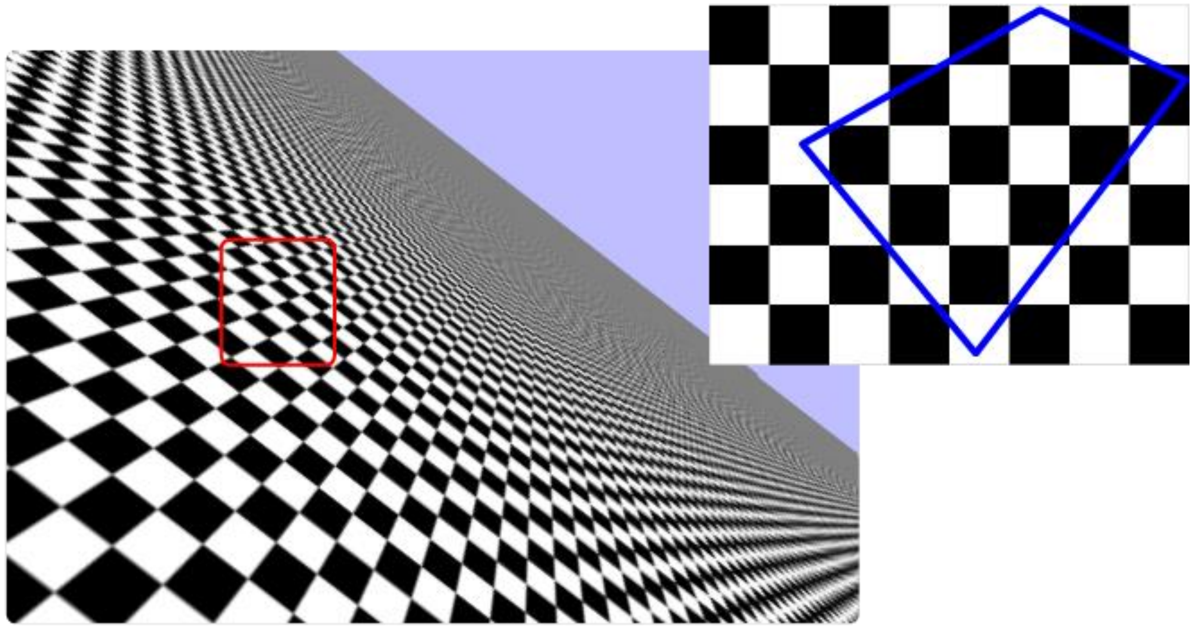
The algorithm chooses the texel that corresponds to a rectangular areas o the original texture, allowing it to obtain a better performance for the cases in which the texture is aligned with camera.



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

Anisotropic

When surfaces are angled with respect to the border of the screen, a pixel on screen corresponds to a generic trapezoid over the texture, which is neither a square, nor a rectangle.



Anisotropic

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

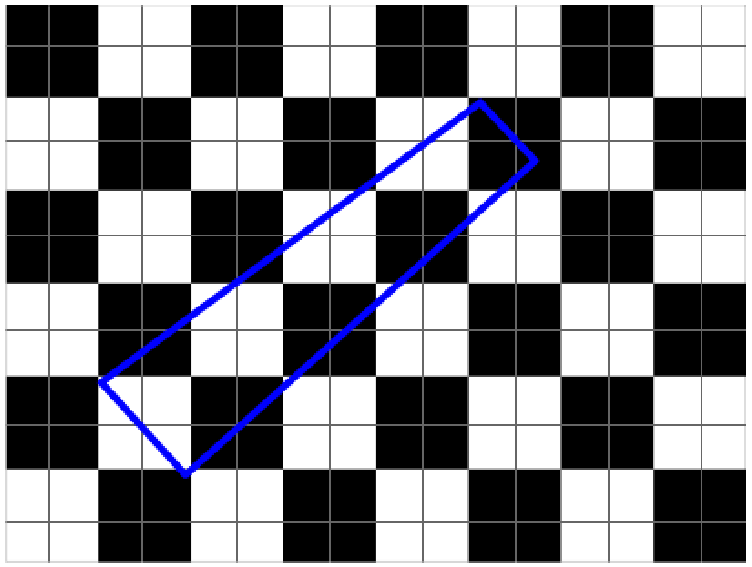
The algorithm approximates the exact area of texels that corresponds to a pixel on screen.

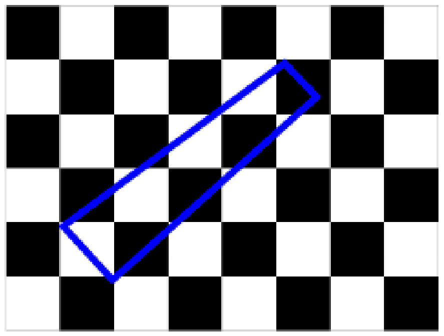
It uses Mip-maps, and samples one of its level following the direction of the pixel over the texels.

Anisotropic

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

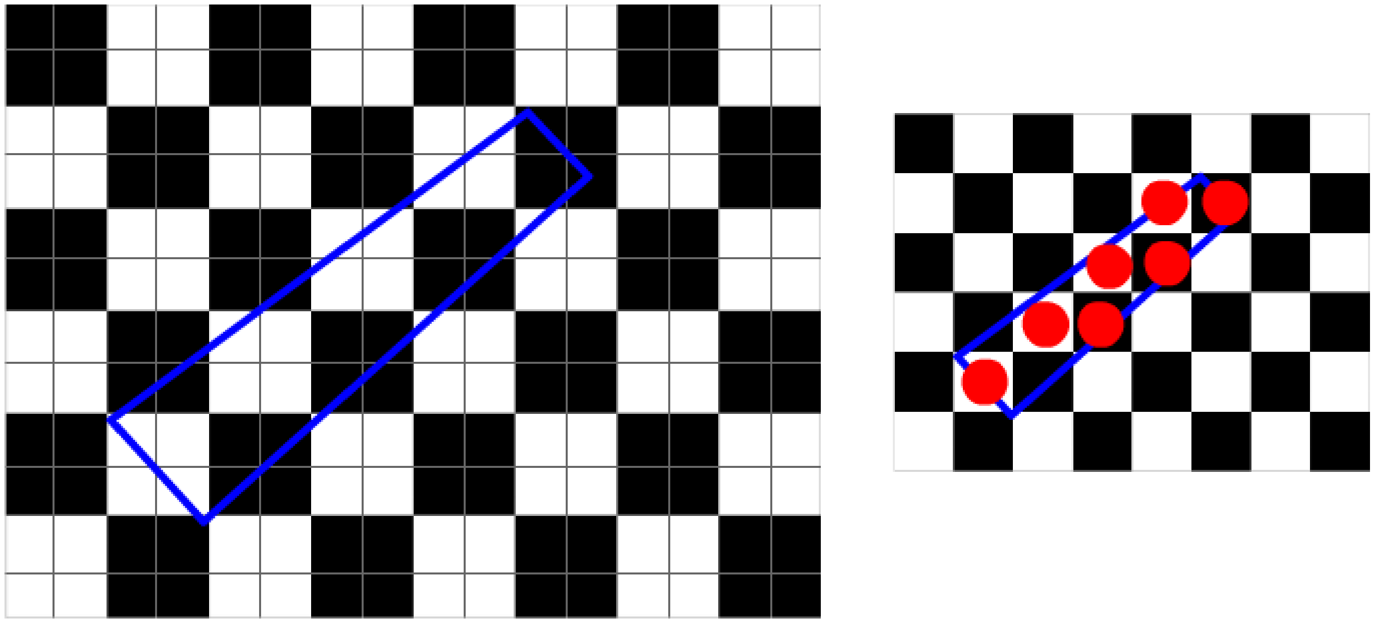
Depending on the size of the smallest side of the trapezoid, the procedure determines the most appropriate Mip-map level (the one with the most similar pixel to texel ratio).





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: for time constraints, we will not enter in details on this feature.

Marco Gribaudo

Associate Professor

CONTACTS

Tel. +39 02 2399 3568
marco.gribaudo@polimi.it
https://www.deib.polimi.it/eng/home-page

(Remember to use the phone, since mails might require a lot of time to be answered. Microsoft Teams messages might also be faster than regular mails)