

Computer Graphics

Ágoston Sipos
siposagoston@inf.elte.hu

Eötvös Loránd University
Faculty of Informatics

2025-2026. Fall semester

Table of contents

Overview

Shading

- Light source models

- Material models

- Implementation

Texturing

- Texture mapping

- Parameterization

- Texture filtering

- Procedural textures

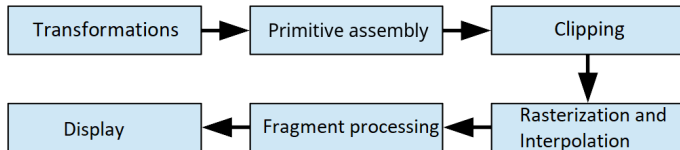
- Non-color textures

Incremental image synthesis on the GPU

- Incremental image synthesis

- Pipeline on hardware

Graphics Pipeline



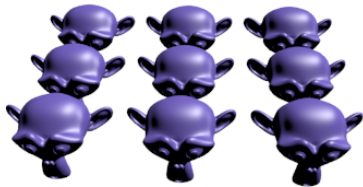
Fragment processing

- ▶ In this step, the color of the fragments will be determined
- ▶ Now we will see how we can do this by modeling light-matter interactions
- ▶ We will also examine what texturing can be used for

Local illumination

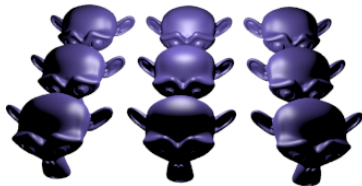
- ▶ We determine the color of the fragment according to some simplified lighting model
- ▶ For this we need
 - ▶ abstract light source models
 - ▶ abstract material models
- ▶ By combining these, we will approximate the desired lighting

Light source models



Directional light source

Light source models



Point light source

Light source models



Spot light source

Abstract light sources

- ▶ Directional light source
 - ▶ it only has direction
 - ▶ represented with one vector (in world CS)
 - ▶ the light rays are considered parallel
 - ▶ used for simulating a distant light source
 - ▶ e.g. sunlight
- ▶ Point light source
 - ▶ it only has a position
 - ▶ represented with one point, in world CS
 - ▶ e.g.: lightbulb
- ▶ *Spot* light source
 - ▶ cone: apex position, axis direction, and a "light circle"
 - ▶ represented with a point, a vector (world CS!) and two angles
 - ▶ e.g.: table lamp

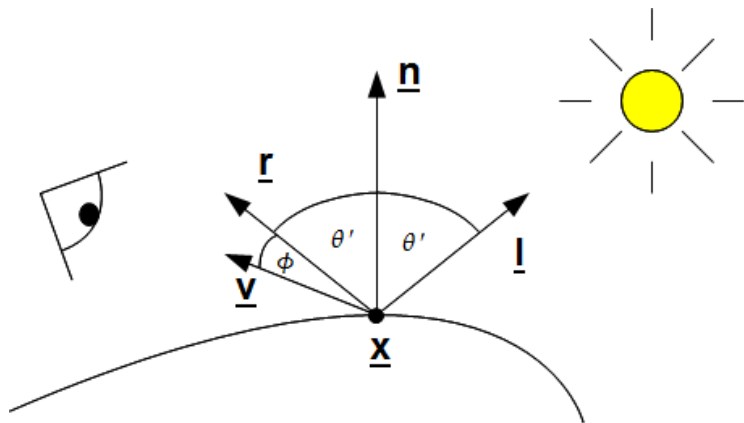
Helmholtz law

- ▶ Helmholtz reciprocity: a ray of light can be reversed
- ▶ This is good for us for two reasons:
 - ▶ It guarantees that ultimately the radiance will decrease.
 - ▶ We can look "backwards" at the rays.

Lighting models – notations

- ▶ $\mathbf{v} := \omega$ viewing direction
- ▶ $\mathbf{l} := \omega'$ towards the light source, vector pointing towards the point "giving" the light
- ▶ \mathbf{n} surface normal
- ▶ \mathbf{r} direction of the ideal reflection from \mathbf{l} incident direction, with \mathbf{n} normal
- ▶ $\mathbf{v}, \mathbf{l}, \mathbf{n}, \mathbf{r}$ unit vectors
- ▶ θ' is the angle between \mathbf{l} and \mathbf{n}
- ▶ ϕ is the angle between \mathbf{r} and \mathbf{v}

Lighting models – notations



BRDF

- ▶ The BDRFs are used to model light-matter interactions
- ▶ Let L^{in} be the intensity of the light coming from a given direction to a point on the surface, and let L be the intensity of the light reflected from there (actually: its radiance, i.e. the power emitted from a unit surface to a unit solid angle)
- ▶ Let \mathbf{l} denote the unit vector towards the light source, \mathbf{v} the unit vector towards the viewpoint, and \mathbf{n} the surface normal at the given point. Let θ' be the angle between \mathbf{n} and \mathbf{l}
- ▶ Then the *bidirectional reflectance distribution function*, BRDF is

$$f_r(\mathbf{x}, \mathbf{v}, \mathbf{l}) = \frac{L}{L^{in} \cos \theta'}$$

- ▶ Note: the $\cos \theta'$ represents the $(-\omega_\ell \cdot \mathbf{n})$ from the global illumination equation (Lecture 6)

Ideal reflection

- ▶ An ideal mirror reflects only in the specular direction \mathbf{r} . (see lecture 4)



$$f_r(\mathbf{x}, \mathbf{v}, \mathbf{l}) = k_r \frac{\delta(\mathbf{r} - \mathbf{v})}{\cos \theta'}$$

- ▶ δ is the *Dirac-delta* function, which is a generalized function whose value is zero everywhere except at zero, but its integral over the real numbers is 1.
- ▶ The k_r reflection coefficient is the *Fresnel coefficient*. This depends on the refractive index of the material and its electrical conductivity.
- ▶ The *Fresnel coefficient* describes the ratio of reflected and incident energy. This physical quantity can be calculated.

Ideal refraction

- ▶ Let \mathbf{t} be the direction of ideal refraction. (see lecture 4)
- ▶ We get it similarly to the ideal reflection:



$$f_r(\mathbf{x}, \mathbf{v}, \mathbf{l}) = k_t \frac{\delta(\mathbf{t} - \mathbf{v})}{\cos \theta'}$$

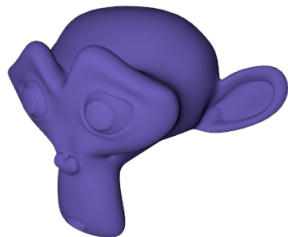
- ▶ k_t is the refraction *Fresnel coefficient*.

Lambert's cosine law

- ▶ Good for describing optically rough, *diffuse* surfaces.
- ▶ Assumption: the amount of reflected light does not depend on the viewing direction.
- ▶ Due to Helmholtz's law, it cannot depend on the incoming direction either, i.e. it is constant:

$$f_r(\mathbf{x}, \mathbf{v}, \mathbf{l}) = k_d$$

- ▶ Only looking at this:
 $L_{ref} = L_i k_d \cos^+ \theta'$



Specular reflection – Phong model

- ▶ It can be used to create a "highlight", that is intensive in the specular direction but quickly fades when moving away from the specular direction
- ▶ Let ϕ be the angle between \mathbf{r} specular direction and \mathbf{v} view direction.
- ▶ Then $\cos \phi = \mathbf{r} \cdot \mathbf{v}$
- ▶ We are looking for a function that is large for $\phi = 0$ but quickly dies off.



$$f_r(\mathbf{x}, \mathbf{v}, \mathbf{l}) = k_s \frac{\cos^n \phi}{\cos \theta'}$$

Not symmetric! So it theoretically violates Helmholtz's law, but in practice it looks nice and is easy to compute.

- ▶ Only looking at this: $L_{ref} = L_i k_s (\cos^+ \phi)^n$

Specular reflection – Phong model



$n = 5$



$n = 25$



$n = 50$

Specular reflection – Phong-Blinn model

- ▶ Let \mathbf{h} be the bisecting vector of the vectors pointing towards the light source and the viewing direction.



$$\mathbf{h} = \frac{\mathbf{v} + \mathbf{l}}{\|\mathbf{v} + \mathbf{l}\|}$$

- ▶ Let δ be the angle between \mathbf{h} and \mathbf{n} normal.
- ▶ Then $\cos \delta = \mathbf{h} \cdot \mathbf{n}$



$$f_r(\mathbf{x}, \mathbf{v}, \mathbf{l}) = k_s \frac{\cos^n \delta}{\cos \theta'}$$

- ▶ Only looking at this: $L_{ref} = L_i k_s (\cos^+ \delta)^n$

Implementation of lighting models

- ▶ The previous formulas describe the behavior of a single wavelength of light.
- ▶ We should describe the entire spectrum, but we simplify: we only use the RGB wavelengths.
- ▶ Most surfaces cannot be described by a single model, we use several models to describe the interaction between material and light.
- ▶ The amount of light from different models is *summed*.
- ▶ Since we only calculate the local illumination, we lose light from multiple reflections. We replace this separately (*ambient light*).

Ambient component

- ▶ The amount of light present everywhere in the scene.
- ▶ Formula: $k_a \cdot L_a$, where "·" is now the multiplication per coordinate: $[a, b, c] \cdot [x, y, z] = [ax, by, cz]$
- ▶ k_a depends on the surface, let
`vec3 ambientColor`
- ▶ Here, k_a determines what fraction of the incident light intensity is reflected by the material at the wavelengths corresponding to R, G, B. k_a is a *surface property*.
- ▶ L_a is the intensity of constant background illumination at RGB wavelengths, independent of light source and surface. L_a is a property of the *scene*.
- ▶ In *fragment shader* (in DirectX pixel shader) can be used:
`ambientColor * ambientLight`

Diffuse component – Lambert law

- ▶ According to the BRDF: $L_{ref} = L_i k_d \cos^+ \theta'$
- ▶ We called this *diffuse color*.
- ▶ k_d and L_i as before, but L_i is the current *light source property*:
`vec3 diffuseColor`
`vec3 diffuseLight`
- ▶ We also need: $\cos^+ \theta'$. (\cos^+ : where \cos is negative, it is 0.)
- ▶ Calculation: `clamp(dot(normal, toLight), 0, 1)`
- ▶ For this you need, `normal = n` and `toLight = l`.
- ▶
$$\text{clamp}(x, a, b) = \begin{cases} a & , \text{if } x < a \\ x & , \text{if } x \in [a, b] \\ b & , \text{if } x > b \end{cases}$$
- ▶ In the case of a *Spot* light source, the light circle will also have to be taken into account.

Specular reflection – Phong model

- ▶ According to the BRDF: $L_{ref} = L_i k_s (\cos^+ \phi)^n$, where ϕ is the angle between \mathbf{r} specular direction and \mathbf{v} viewing direction.
- ▶ k_d and L_i (this is a different L_i) again as before, L_i is also a the current light's *light source property*:
`vec3 specularColor`
`vec3 specularLight`
- ▶ n is a surface-dependent constant, let it be
`float specularPower`

Specular reflection – Phong model

$L_{ref} = L_i k_s (\cos^+ \phi)^n$; \mathbf{r} specular direction and \mathbf{v} viewing direction.

- ▶ We need $\cos^+ \phi$, for which we need \mathbf{r} and \mathbf{v} .
- ▶ \mathbf{r} is the mirror image of \mathbf{l} vector on \mathbf{n} . Calculation:
`vec3 refl = reflect(-toLight, normal)`
- ▶ \mathbf{v} is the viewing direction, i.e. the unit vector pointing from the surface point to the camera.
`vec3 toEye = normalize(eyePosition - worldPos)`
- ▶ Calculating $(\cos^+ \phi)^n$:
`pow(clamp(dot(refl, toEye), 0, 1), specularPower)`

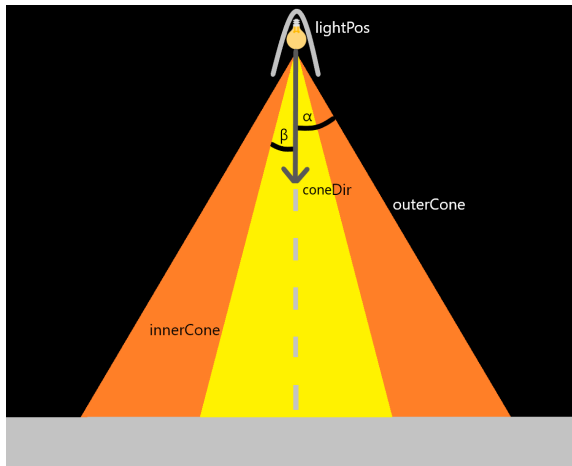
Calculating toLight

- ▶ Directional light source
 - ▶ Direction of the light, normalized direction vector: `vec3 lightDirection`
 - ▶ `toLight = -lightDirection`
- ▶ Point light source
 - ▶ Light's position, position vector: `vec3 lightPosition`
 - ▶ `toLight = normalize(lightPosition - worldPos)`
- ▶ *Spot* light source
 - ▶ Axis of the lighting cone, normalized direction vector: `vec3 coneDirection`
 - ▶ Light's position, position vector: `vec3 lightPosition`
 - ▶ `toLight = normalize(lightPosition - worldPos)`
like a point light source

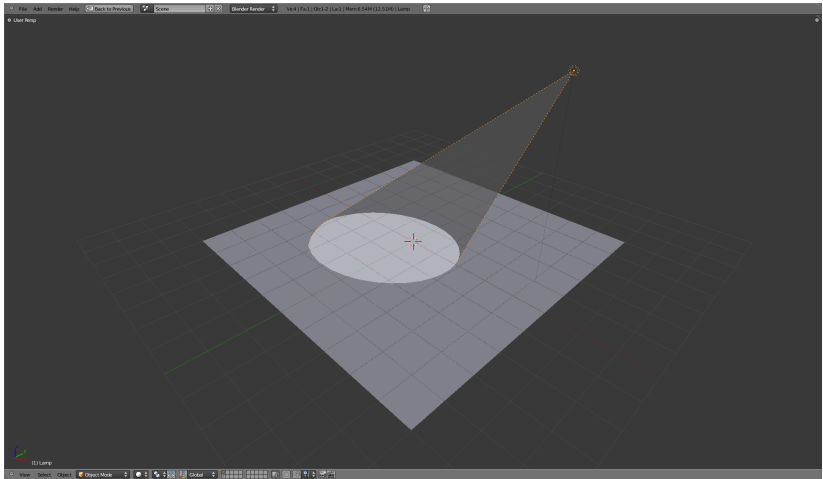
Effect of *spot* light source

- ▶ Two extra parameters:
 - ▶ inner light circle: within which it acts with full intensity
 - ▶ outer light circle: outside of which it has absolutely no effect
- ▶ Between the two, the light intensity decreases
- ▶ The light circles are considered to be (infinite) cones starting from the light source and having the same orientation as the direction of the light.
- ▶ A surface point is inside a cone if the line connecting the point to the position of the light source is inside the cone.

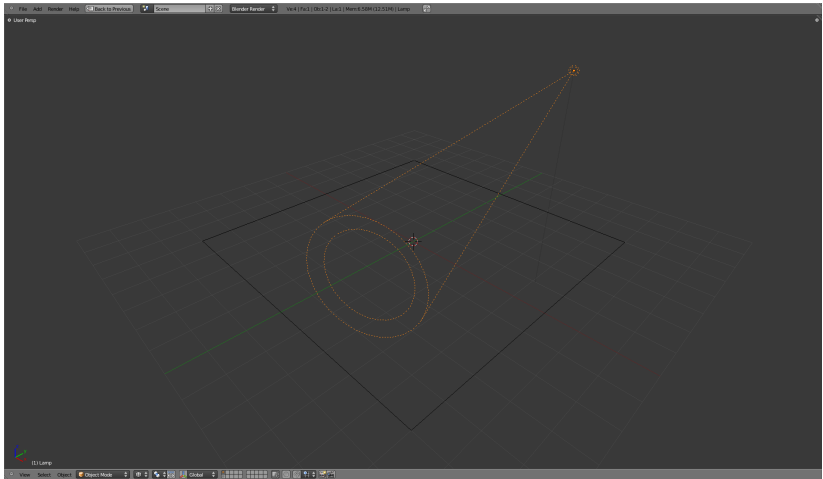
Spot light source



Spot light source



Spot light source



Effect of *spot* light source

- ▶ If the angle between `coneDirection` and `-toLight` is less than half of the opening angle of the cone, then the segment is inside the cone, i.e. the surface point is inside too
- ▶ Instead of the angles, it is sufficient to examine their cos if the max angle is $\leq 180^\circ$
- ▶ Let's represent the light circles, with cos of the half angle to the corresponding cone:
 - ▶ inner light-circle: `cosInnerCone`
 - ▶ outer light-circle: `cosOuterCone`

Effect of *spot* light source

- ▶ `smoothstep(min, max, x):`
 - ▶ 0, if $x < \min$
 - ▶ 1, if $x > \max$
 - ▶ a transition between 0 and 1 (cubic Hermite)
- ▶ `float spotFactor = smoothstep(cosOuterCone, cosInnerCone, dot(coneDirection, -toLight))`
- ▶ The diffuse and specular terms must be multiplied with this

In summary

Surface properties

- ▶ ambientColor
- ▶ diffuseColor
- ▶ specularColor
- ▶ specularPower
- ▶ normal
- ▶ worldPos

Light source properties

- ▶ diffuseLight
- ▶ specularLight
- ▶ toLight

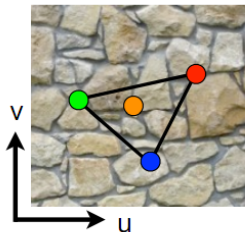
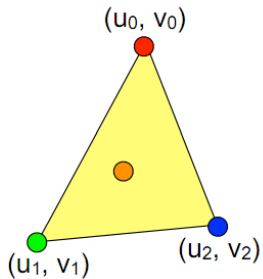
Scene properties

- ▶ ambientLight
- ▶ eyePosition

What is texturing?

- ▶ So far: *single color* material models, i.e. the entire surface is the same color.
- ▶ Not common in the real world.
- ▶ We want to give fine details.
- ▶ We need different parameters in BRDF.
- ▶ We specify these parameters – mainly color – in the *textures*.

Texturing



Texture description methods

- ▶ "Array":
 - ▶ we read from some 1/2/3 dimensional array, and its elements are *texels*
 - ▶ Visually in 2D: we "cover/wrap" our geometry with the "image" stored in the array
 - ▶ We identify the texels with coordinates in *texture space*, the coordinates are interpreted on the $[0, 1]$ interval
- ▶ We give it as a function:
 - ▶ using some $f: (x, y, z) \rightarrow \text{color}$ or $f: (u, v) \rightarrow \text{color}$ function
 - ▶ this is called *procedural texturing*

Mapping methods

- ▶ We have two spaces: *image space* (screen's pixels) and *texture space* (texture's texels)
- ▶ From where to where do we map?
- ▶ Texture space \rightarrow image space: *texture based mapping*
 - » For each *texel*, we search for its corresponding *pixel*.
 - ⊕ Efficient.
 - ⊖ It is not guaranteed that we "hit" every pixel, if we hit a pixel we might hit it multiple times.
- ▶ Image space \rightarrow texture space: *image space based mapping*
 - » For each *pixel*, we search for its corresponding *texel*.
 - ⊕ It fits well with incremental image synthesis.
 - ⊖ It requires the inverse of the parameterization and projection transformations.

Parameterization

- ▶ How can we decide for each surface point, which elements of the array are needed or with which parameters to evaluate the procedural texture function?
- ▶ *texture coordinates* are assigned to each point of the surface.
- ▶ The assignment of texture coordinates to the surface is now called *parameterization*.
- ▶ In the following, we will talk about 2D textures.

Parameterizing parametric surfaces

- ▶ Parametric surface:

$$F \in \mathbb{R}^2 \rightarrow \mathbb{R}^3, \quad F(u, v) := (x, y, z)$$

- ▶ naturally the u, v parameters can be used as texture coordinates.
- ▶ If $\mathcal{D}_F \neq \mathcal{D}_{tex}$, then we need to transform u, v .
- ▶ E.g.:
 - ▶ Cylindrical surface
 - ▶ $\mathcal{D}_F = [0, 2\pi] \times [0, h], F(u, v) := (\cos u, v, \sin u)$
 - ▶ Texture space: $(\bar{u}, \bar{v}) \in [0, 1] \times [0, 1]$
 - ▶ Transformation: $\bar{u} := u/2\pi, \bar{v} := v/h$

Parameterizing triangles

- ▶ Let triangle vertices: $p_i = (x_i, y_i, z_i) \in \mathbb{R}^3$, $i \in \{1, 2, 3\}$, as well as their corresponding vertices in texture space: $t_i = (u_i, v_i) \in \mathbb{R}^2$, $i \in \{1, 2, 3\}$.
- ▶ We are looking for a mapping $\mathbb{R}^3 \rightarrow \mathbb{R}^2$ such that $p_i \mapsto t_i$ and it maps a triangle to triangle.
- ▶ The simplest such mapping is the linear mapping, which can be given by a 3×3 matrix.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} A_x & A_y & A_z \\ B_x & B_y & B_z \\ C_x & C_y & C_z \end{bmatrix} \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{P} \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

Parameterizing triangles

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} A_x & A_y & A_z \\ B_x & B_y & B_z \\ C_x & C_y & C_z \end{bmatrix} \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{P} \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

- ▶ Nine unknowns, nine equations
- ▶ Texture based mapping!
- ▶ For screen based mapping we need the inverse transformation:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \mathbf{P}^{-1} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Texturing and ray tracing

- ▶ Surface-ray intersection: in world coordinate system
- ▶ Inverse transformations:
 - ▶ World CS \rightarrow Model CS
 - ▶ Model CS \rightarrow texture space

Model CS \rightarrow texture space

- ▶ Parametric surface

We get u, v during intersection calculations, it does not need to be calculated separately.

- ▶ Triangles

The previously derived \mathbf{P}^{-1} required.

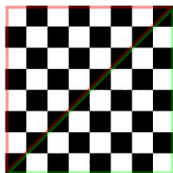
Speed up option: if neither the triangle itself nor the texture coordinates change in the vertices, then \mathbf{P}^{-1} is constant.

Parameterizing triangles

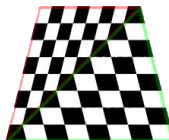
- ▶ In practice, this can be calculated quickly with an incremental algorithm.
- ▶ If the texture coordinates are given in the three vertices, they can be calculated for each pixel with the algorithm used for filling the triangles.
- ▶ Let α, β, γ be the barycentric coordinates of the points on the surface where $\mathbf{p} = \alpha\mathbf{p}_1 + \beta\mathbf{p}_2 + \gamma\mathbf{p}_3$
- ▶ Then we get the texture coordinate for \mathbf{p} with $t = \alpha t_1 + \beta t_2 + \gamma t_3$

Perspective-correct texture mapping

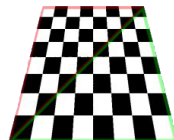
- ▶ The linear interpolation of texture coordinates will give the wrong image, if we are not using only affine transformations on the triangles.
- ▶ I.e: 99% of the time.



Plane

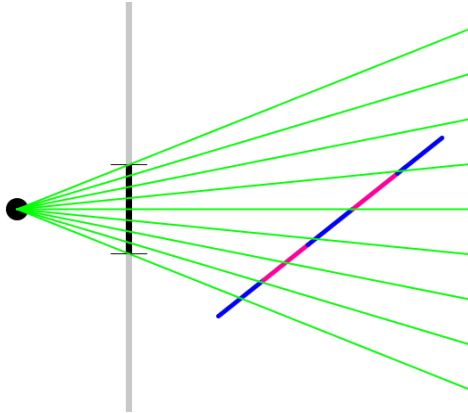


Affine



Correct

Perspective-correct texturing



Perspective-correct texturing

- ▶ So linear interpolation of u, v won't be good for non-affine transformations – because they are not linear in screen space
- ▶ After we transform, but before homogeneous divide, our coordinates should be $[x_t, y_t, z_t, w_t]$
- ▶ After homogeneous divide: $[x_s, y_s, z_s, 1] = [\frac{x_t}{w_t}, \frac{y_t}{w_t}, \frac{z_t}{w_t}, 1]$
- ▶ If we interpolate the texture coordinates based on $[x_s, y_s, z_s, 1]$, then it won't be correct.
- ▶ In contrast to that: if we interpolate $\frac{1}{w}$, it will remain correct! Moreover, any $\frac{q}{w}$ value is well interpolated!
- ▶ Instead of α, β, γ we use the coordinates $\alpha_w, \beta_w, \gamma_w$ in world CS, and
- ▶ interpolating $\frac{\alpha_w}{w}, \frac{\beta_w}{w}, \frac{\gamma_w}{w}$, we get the correct texturing.

*Perspective-correct texturing - proof 1/3

- ▶ Now we denote the screen coordinates as X, Y and the spatial coordinates as x, y, z , where the relation between the two:

$$X = \frac{x}{z}, \quad Y = \frac{y}{z}$$

from which obviously $x = Xz$, $y = Yz$.

- ▶ Let us assume that the point above is a vertex of a triangle (or polygon in a general case) and let the implicit equation of this primitive's plane be

$$Ax + By + Cz = D$$

- ▶ Let u, v denote the texture coordinates of the spatial primitive points, which we assume can be expressed as a linear function of the x, y, z coordinates:

$$u = ax + by + cz + d, \quad v = ex + fy + gz + h$$

*Perspective-correct texturing - proof 2/3

Let's show that $1/z$ is a linear function of X, Y :

- ▶ Let's start from the implicit equation of our primitive's plane:

$$Ax + By + Cz = D$$

when we substitute x and y with the screen coordinates (X, Y) we get the following expression:

$$AXz + BYz + Cz = D$$

- ▶ Divide this by zD (i.e. express $1/z$):

$$\frac{1}{z} = \frac{A}{D}X + \frac{B}{D}Y + \frac{C}{D}$$

- ▶ Since A, B, C, D are constants (i.e. independent of X, Y), the above expression is indeed the $1/z$ as a linear function of X, Y

*Perspective-correct texturing - proof 3/3

Let's show that $u/z, v/z$ is a linear function of X, Y :

- We assumed that texture coordinates are linear in spatial coordinates, i.e

$$u = ax + by + cz + d$$

$$v = ex + fy + gz + h$$

by substituting the spatial coordinates with the screen coordinates, we get this expression

$$u = aXz + bYz + cz + d$$

$$v = eXz + fYz + gz + h$$

*Perspective-correct texturing - proof 3/3

Let's show that $u/z, v/z$ is a linear function of X, Y :

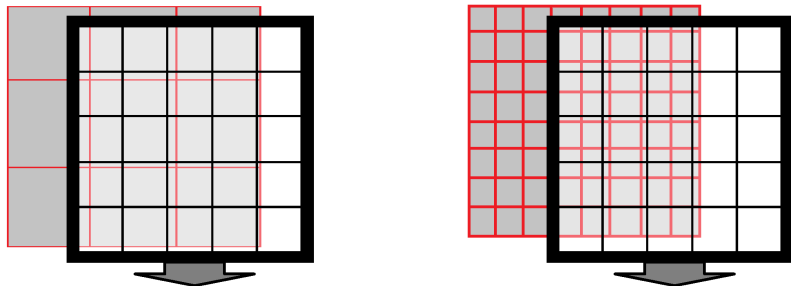
- ▶ Divide by z and substitute the $1/z$ on the right side with the previously obtained expression:

$$\frac{u}{z} = aX + bY + c + d \left(\frac{A}{D}X + \frac{B}{D}Y + \frac{C}{D} \right)$$
$$\frac{v}{z} = eX + fY + g + h \left(\frac{A}{D}X + \frac{B}{D}Y + \frac{C}{D} \right)$$

and they are indeed linear, because by rearranging the above

$$\frac{u}{z} = \left(a + d\frac{A}{D} \right) X + \left(b + d\frac{B}{D} \right) Y + \left(c + d\frac{C}{D} \right)$$
$$\frac{v}{z} = \left(e + h\frac{A}{D} \right) X + \left(f + h\frac{B}{D} \right) Y + \left(g + h\frac{C}{D} \right)$$

Texture filtering



- ▶ It is rare to have exactly one texel per pixel.
- ▶ Magnification: the pixel size is smaller than the texel size – multiple pixel per texel. OpenGL: `GL_TEXTURE_MAG_FILTER`
- ▶ Minifying: the pixel size is larger than the texel size – multiple texel per pixel. OpenGL: `GL_TEXTURE_MIN_FILTER`
- ▶ Another problem: if something is linear in texture space, then it's not linear in screen space because of the perspective

Magnification

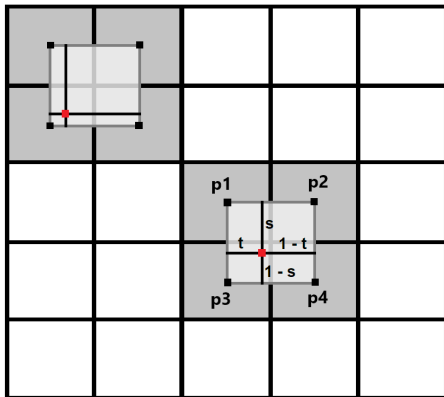
The pixel size is smaller than the texel size – multiple pixel per texel

- ▶ Nearest neighbour: we use the value of the texel closest to the pixel's center
- ▶ Bilinear filtering: we take the weighted average of the nearest four texels.

Bilinear filtering

$$\mathbf{b}(s, t) = (1 - t)((1 - s)\mathbf{p}_1 + s\mathbf{p}_3) + t((1 - s)\mathbf{p}_2 + s\mathbf{p}_4)$$

$$s, t \in [0, 1]$$



Minifying

The pixel size is larger than the texel size – multiple texel per pixel

- ▶ The accurate sampling would be: transform the square of the pixel into texture space and take the average of the selected texels.
- ▶ Instead: we also take a square in the texture space.
- ▶ In practice: averaging of an arbitrary square of the texture is too resource demanding, instead we should either use fewer texels or *MIP maps*
- ▶ Fewer texels:
 - ▶ Nearest neighbour: we use the value of the texel closest to the pixel's center
 - ▶ Bilinear filtering: we take the weighted average of the nearest four texels.

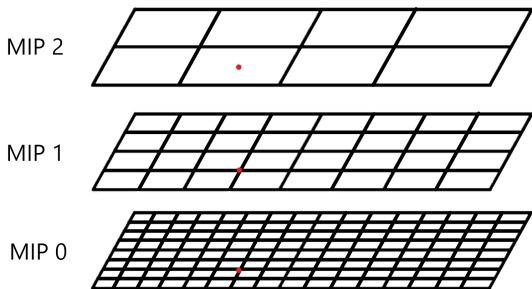
MIP-maps

- ▶ MIP: *multum in parvo* – many things in a small place
- ▶ We generate a "pyramid" from the texture, halving its size at each level



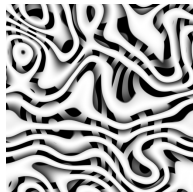
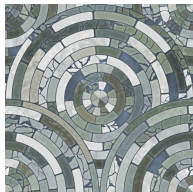
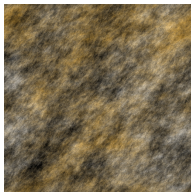
MIP-maps

- ▶ During filtering, we select the appropriate level based on the ratio of pixel/texel area and read from there.
- ▶ Even within the given MIP-map, reading can be unfiltered or bilinearly filtered.
- ▶ *Trilinear filtering*: we use two adjacent levels, we use bilinear filtering within the levels, then we take the weighted average of the results



Procedural textures

- ▶ The textures can be specified with a function instead of an "array".
- ▶ Texture coordinates: the function parameters.



generated with Filter Forge

Properties

- ▶ Advantages:

- ▶ A lot less storage space is required
- ▶ Any resolution – the only limit is numerical accuracy

⇒ In case of magnification we don't need to filter.

(Unfortunately, it does not solve the minification problem)

- ▶ Drawbacks:

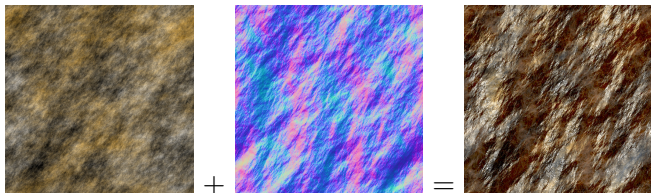
- ▶ High computational demand.
- ▶ It can be harder to modify it.

Non-color textures

- ▶ Textures can be used to describe any property of a surface point.
- ▶ These properties can be for example:
 - ▶ surface normal – *bump and normal mapping*
 - ▶ displacement – *Displacement mapping*
 - ▶ light source visibility – *Shadowmaps*
 - ▶ mirror like reflection – *Reflection mapping/Environment mapping*

Bump or normal mapping

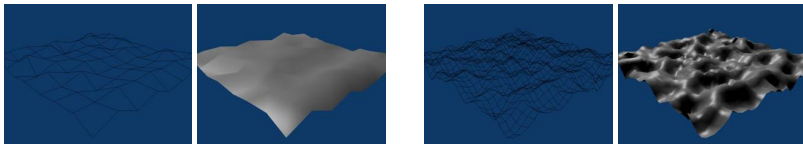
- ▶ With the texture we specify normal vectors.
- ▶ Instead of the original normals of the surface, we use them when calculating the lighting.
- ▶ It gives the appearance of a bumpy/rough surface until you look at it, at a very flat angle.



generated with Filter Forge

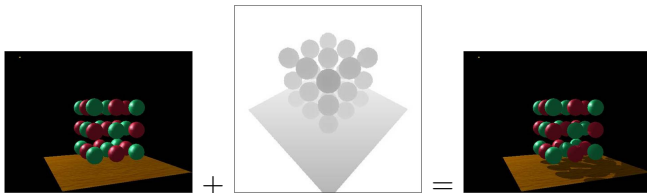
Displacement mapping

- ▶ The surface point is actually moved
- ▶ We can only move the vertices of triangles \Rightarrow depends on the resolution of the geometry.
- ▶ Even at a flat angle, we get a good result.



Shadowmaps

- ▶ From the light source's point of view, we create a texture in which the distances from the light source are stored.
- ▶ During the actual drawing, based on the shadowmap, we decide for each point whether it is directly affected by the light or not.

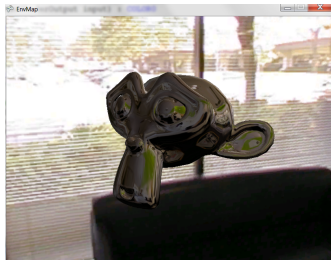
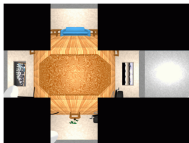
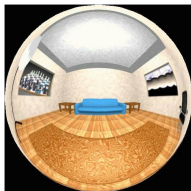


Reflection mapping

- ▶ Can be used for flat mirrors.
- ▶ We create a separate image, saved in a texture, of what is visible in the reflected direction.
- ▶ This will be applied to the surface as a texture during the final drawing.
- ▶ It only gives a single bounce/reflection.
- ▶ It must be done separately for each mirror.

Environment mapping

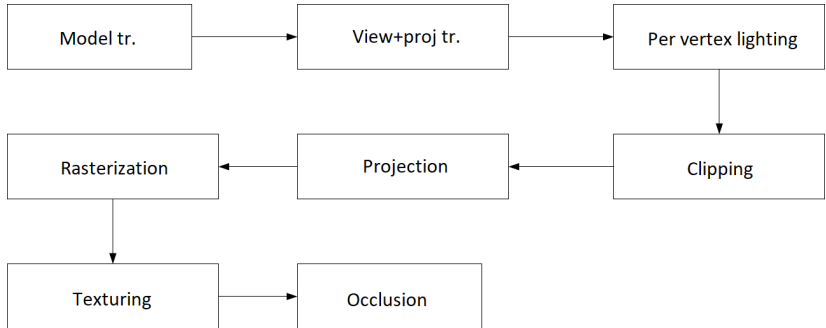
- ▶ We consider the environment of our scene to be infinitely distant, in the query only the direction matters, not the position.
- ▶ We store it in a special texture. (Typically *Cubemap*.)
- ▶ When drawing, for reflective surfaces, we read from this according to the direction of the reflection.



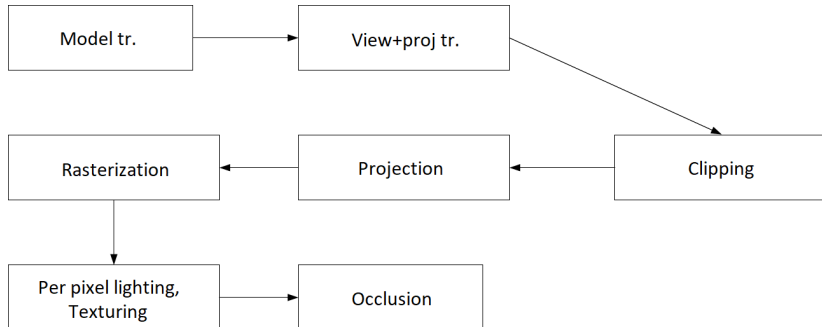
In summary

- ▶ All surface-optical properties can be specified with a constant or even with a texture.
- ▶ (Even more things if we are clever!)
- ▶ All vectors and points are given in the world coordinate system.
- ▶ `eyePosition` must be updated when the view changes!
- ▶ Problem: our model is in model CS, and the *Vertex Shader* will transfer it to normalized device CS!
- ▶ Solution: let's also calculate the world coordinates with the *Vertex Shader*, and pass the interpolated value to the *Fragment Shader*!

Incremental image synthesis



Incremental image synthesis



Pipeline

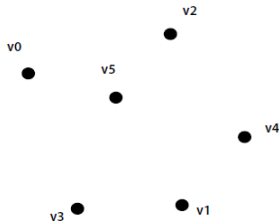
- ▶ We divide the work into subtasks
- ▶ Each subtask is processed by a different processing unit (processor) (ideally)
- ▶ The input of each unit is the output of the unit preceding it in the pipeline

Pipeline

- ▶ In order for a unit to start working, it does not have to wait for work on the *entire* workpiece to be completed, only the parts in front of it need to be ready
- ▶ → if we introduce n pipeline **stage**, then at a given moment we can work on up to n elements at the same time (after the initial start-up)
- ▶ Not just GPU! Even the Pentium IV had 20 pipeline stages (a GeForce 3 has 6-800)

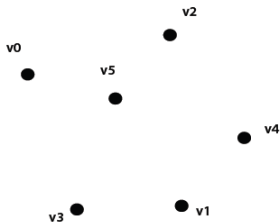
Parallelizations in practice – per vertex

- ▶ When we call `glDrawArrays`, `glDrawElements` etc., then the pipeline starts on the vertices of the primitive
- ▶ We transform each **vertex** *independently, in parallel*

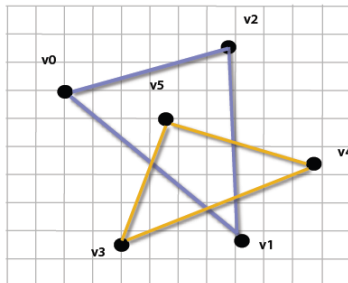


Parallelizations in practice – per primitive

- ▶ All **primitives** to be drawn are clipped *independently, in parallel*



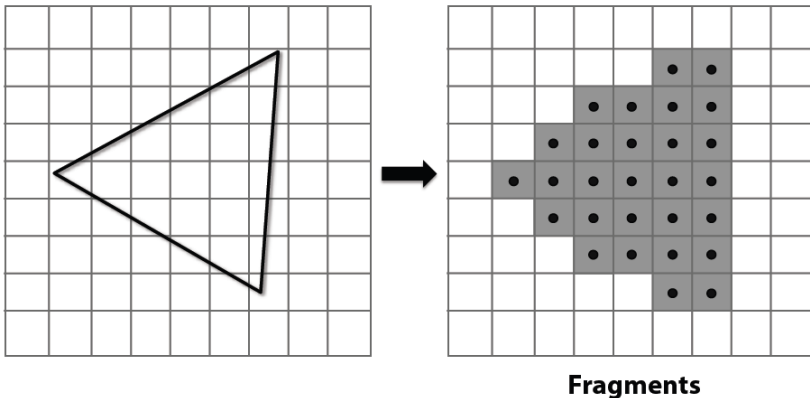
Vertices



**Primitives
(triangles)**

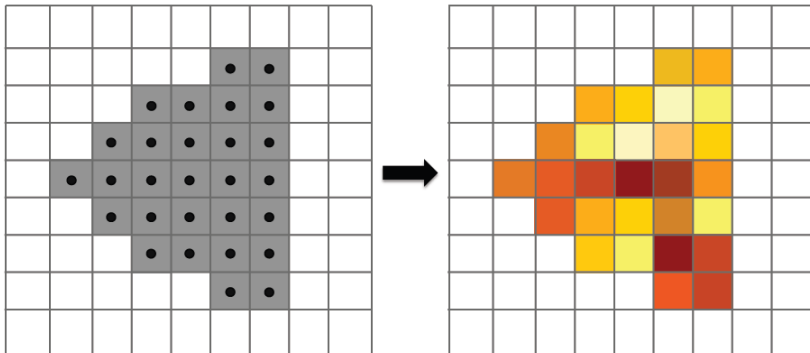
Parallelizations in practice – per primitive

- ▶ All **primitives** to be drawn are rasterized *independently, in parallel*



Parallelizations in practice – per fragment

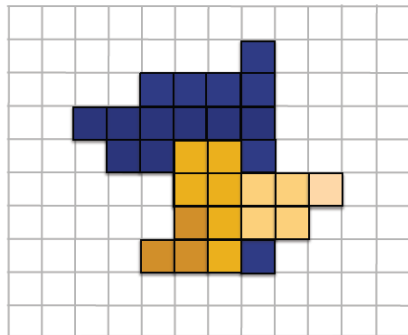
- ▶ All **fragments** to be drawn are colored *independently, in parallel*



Shaded fragments

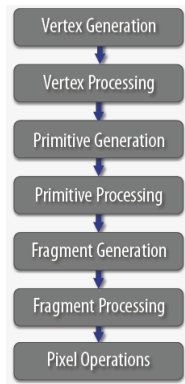
Parallelizations in practice – per pixel

- For each **pixel** corresponding to a fragment, we define the color of the pixel displayed on the screen (+visibility with z-buffer)



Pixels

Pipeline on hardware



GPU programming

- ▶ No explicit parallelization
- ▶ Because we get that from working on different elements independently, in parallel (even in time)

Compiling shaders

```
sampler mySamp;  
Texture2D<float3> myTex;  
float3 lightDir;  
  
float4 diffuseShader(float3 norm, float2 uv)  
{  
    float3 kd;  
    kd = myTex.Sample(mySamp, uv);  
    kd *= clamp( dot(lightDir, norm), 0.0, 1.0);  
    return float4(kd, 1.0);  
}
```

1 input fragment record



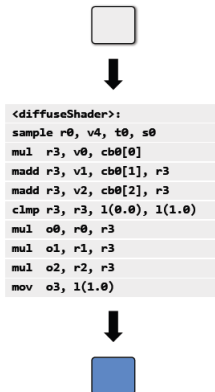
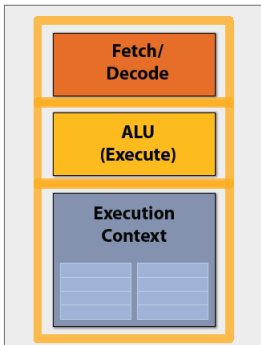
```
<diffuseShader>:  
sample r0, v4, t0, s0  
mul r3, v0, cb0[0]  
madd r3, v1, cb0[1], r3  
madd r3, v2, cb0[2], r3  
clmp r3, r3, 1(0.0), 1(1.0)  
mul o0, r0, r3  
mul o1, r1, r3  
mul o2, r2, r3  
mov o3, 1(1.0)
```



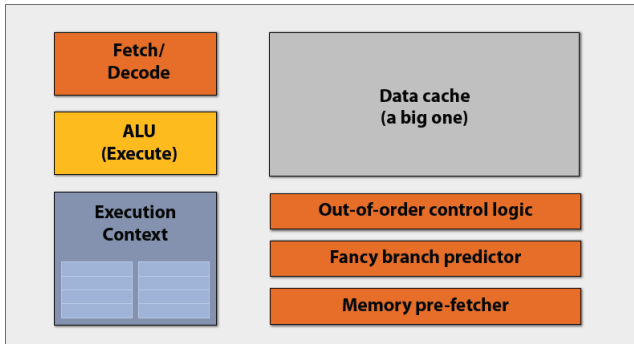
1 output fragment record



Executing shaders

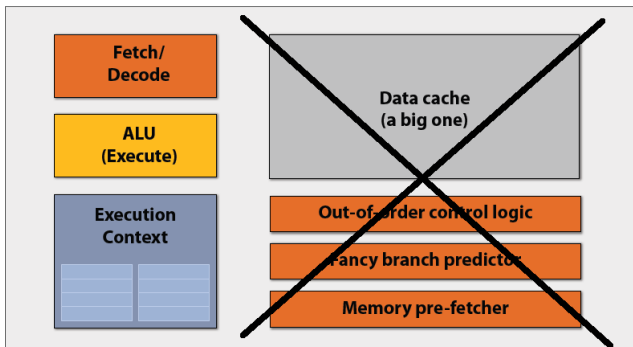


A typical CPU



Processing unit of a GPU

Remove the components that help with fast execution of only one thread

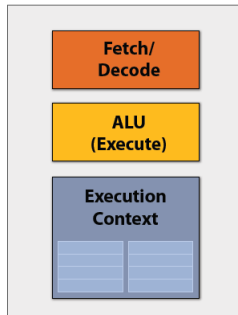
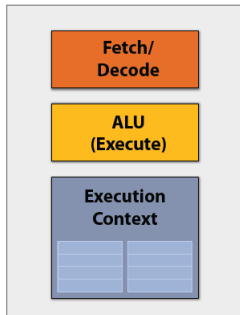


GPU – 2 processing unit, 2 shader

fragment 1



```
<diffuseshader>:  
sample r0, v0, t0, s0  
mul r3, v0, cb0[0]  
madd r3, v1, cb0[1], r3  
madd r3, v2, cb0[2], r3  
clap r3, r3, l(0.0), l(1.0)  
mul o0, r0, r3  
mul o1, r1, r3  
mul o2, r2, r3  
mov o3, l(1.0)
```



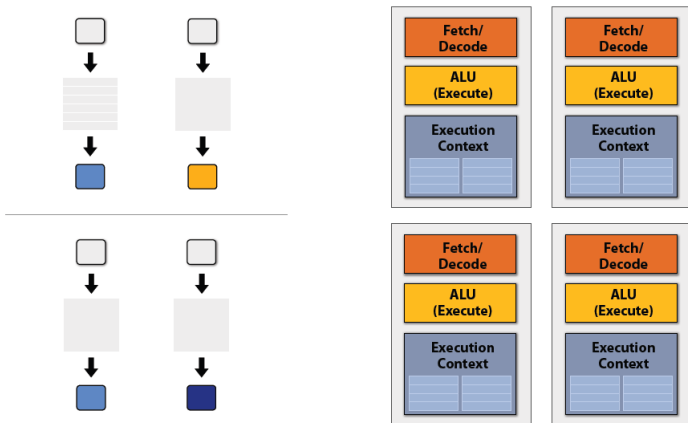
fragment 2



```
<diffuseshader>:  
sample r0, v0, t0, s0  
mul r3, v0, cb0[0]  
madd r3, v1, cb0[1], r3  
madd r3, v2, cb0[2], r3  
clap r3, r3, l(0.0), l(1.0)  
mul o0, r0, r3  
mul o1, r1, r3  
mul o2, r2, r3  
mov o3, l(1.0)
```



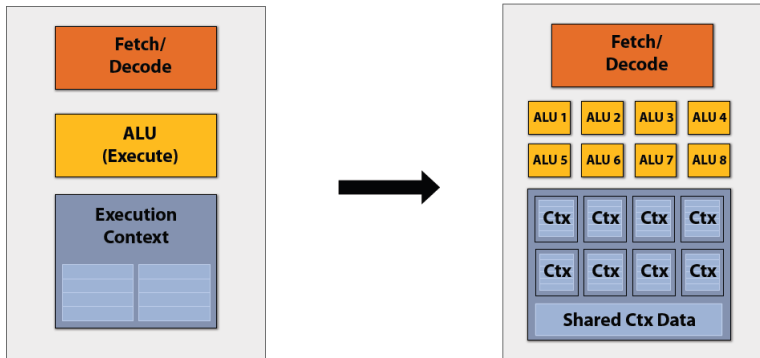
GPU – 4 processing unit, 4 shader



GPU processing units

- ▶ In the previous figures, each processing unit ran a different shader
- ▶ But this is not necessary!
- ▶ A triangle can be made into many fragments → the same shader must be run for each!
- ▶ ⇒ we reduce costs by assigning a common instruction fetcher to several ALUs (i.e they execute the same program) → **Single Instruction Multiple Data**

GPU processing unit



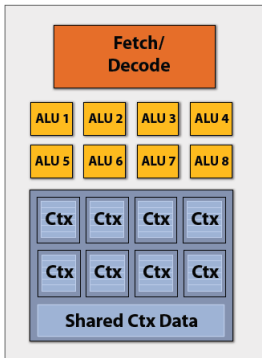
GPU processing unit

```
<diffuseShader>:  
sample r0, v4, t0, s0  
mul  r3, v0, cb0[0]  
madd r3, v1, cb0[1], r3  
madd r3, v2, cb0[2], r3  
clmp r3, r3, 1(0.0), 1(1.0)  
mul  o0, r0, r3  
mul  o1, r1, r3  
mul  o2, r2, r3  
mov  o3, 1(1.0)
```



```
<VEC8_diffuseShader>:  
VEC8_sample vec_r0, vec_v4, t0, vec_s0  
VEC8_mul    vec_r3, vec_v0, cb0[0]  
VEC8_madd   vec_r3, vec_v1, cb0[1], vec_r3  
VEC8_madd   vec_r3, vec_v2, cb0[2], vec_r3  
VEC8_clmp   vec_r3, vec_r3, 1(0.0), 1(1.0)  
VEC8_mul    vec_o0, vec_r0, vec_r3  
VEC8_mul    vec_o1, vec_r1, vec_r3  
VEC8_mul    vec_o2, vec_r2, vec_r3  
VEC8_mov    o3, 1(1.0)
```

GPU processing unit

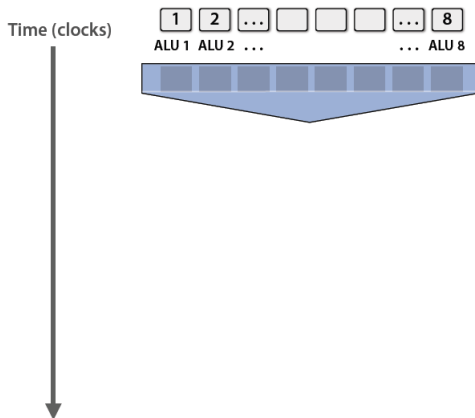


<VEC8_diffuseShader>:

```
VEC8_sample vec_r0, vec_v4, t0, vec_s0  
VEC8_mul vec_r3, vec_v0, cb0[0]  
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3  
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3  
VEC8_clmp vec_r3, vec_r3, 1(0.0), 1(1.0)  
VEC8_mul vec_o0, vec_r0, vec_r3  
VEC8_mul vec_o1, vec_r1, vec_r3  
VEC8_mul vec_o2, vec_r2, vec_r3  
VEC8_mov o3, 1(1.0)
```



Branching

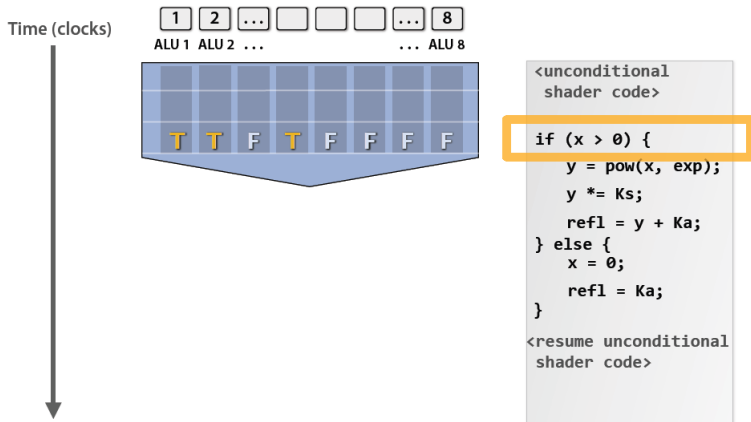


```
<unconditional  
shader code>
```

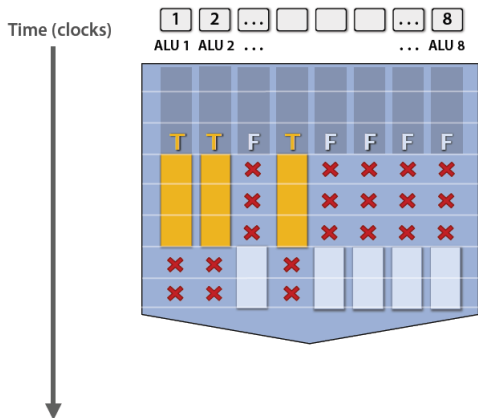
```
if (x > 0) {  
    y = pow(x, exp);  
    y *= Ks;  
    refl = y + Ka;  
} else {  
    x = 0;  
    refl = Ka;  
}
```

```
<resume unconditional  
shader code>
```

Branching



Branching



<unconditional
shader code>

```
if (x > 0) {
```

```
    y = pow(x, exp);
```

```
    y *= Ks;
```

```
    refl = y + Ka;
```

```
} else {
```

```
    x = 0;
```

```
    refl = Ka;
```

```
}
```

<resume unconditional
shader code>

Branching

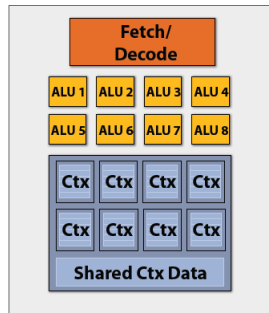
- ▶ The ALU1-ALU8 denotes the ALUs with shared instruction fetcher
- ▶ T appeared for those ALUs whose data set the branching condition to *true*
- ▶ And F for those with *false*
- ▶ The instructions of both branches (!) must be evaluated
- ▶ In the worst case, the performance in the figure above drops to one eighth!
- ▶ Today's GPUs have 16-64 ALUs on a shared instruction stream

Stalls

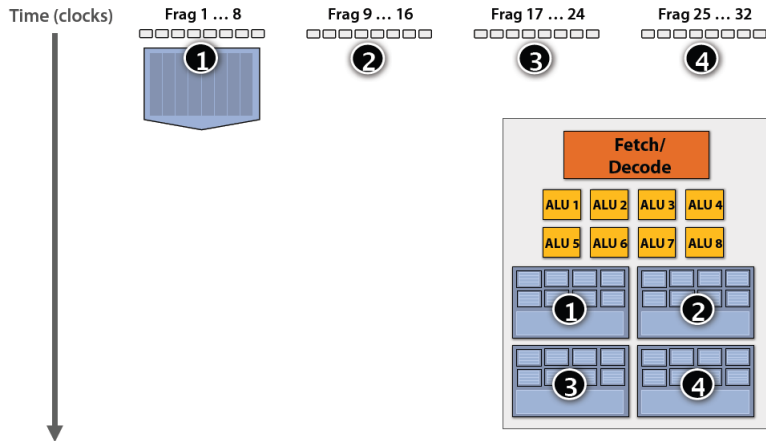
- ▶ On one ALU with multi-threading we process multiple fragment/vertex/etc. → storing multiple execution contexts (code, temporal data, etc.)
- ▶ Because some operations are much slower than others
- ▶ For example: a texture query in a shader is 100x or even 1000x slower than an arithmetic (vector) operation!

Paging

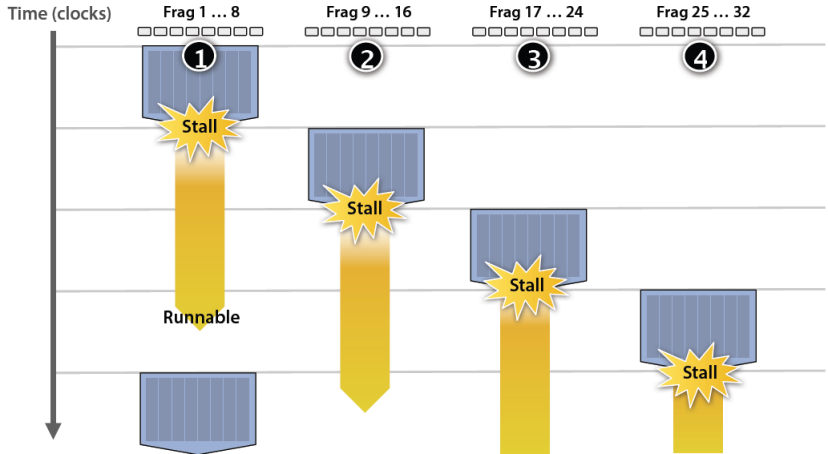
Time (clocks)



Paging



Paging



GPU processing unit

The following observations played an important role in the construction of GPU stream processors:

1. Remove the components that help with fast execution of only one thread
2. \Rightarrow we reduce costs by assigning a shared instruction fetcher to several ALUs (i.e they execute the same program) \rightarrow **Single Instruction Multiple Data**
3. On one ALU with multi-threading we process multiple fragment/vertex/etc. \rightarrow storing multiple execution contexts (code, temporal data, etc.)