Computer Graphics

Ágoston Sipos
siposagoston@inf.elte.hu

Eötvös Loránd University
Faculty of Informatics

2025-2026. Fall semester

Table of contents

Reminder

- Last lecture we took a look at recursive ray tracing
- Advantages:
  - The scene can be populated with anything, that can be intersected with a ray
  - Easy to implement with recursion
  - It treats light as a particle, the resulting phenomena can be easily displayed
- At the same time, we have seen that it also has disadvantages:
  - For every pixel we need to test every primitive $\rightarrow$ this is what we tried to speed up (bounding, space partitioning)
  - The algorithm is global in nature thus difficult to accelerate with hardware
  - It cannot reproduce the phenomena arising from the wave nature of the light
  - Too slow for real-time applications

Raycasting

```
For every pixel on the screen:
    For every object in the scene:
        Does the ray hit the object?
```

Incremental image synthesis

```
For every object in the scene:
    For every pixel on the screen:
        Is the pixel covered by the object's projection?
```

Real-time graphics

- For ray tracing, it was: $\forall$ pixel start a ray: $\forall$ object (geometry) check if there is an intersection
- Instead let's try this: for $\forall$ object (geometry): calculate which pixels it is mapped to and finally display only the closest one!
- The speed depends a lot on how quickly can we decide whether a pixel is covered by the object or not.
- For this reason, objects can only be built from simple geometries $\Rightarrow$ in practice this means linear elements (segments and triangles)
- All other geometries (e.g. sphere) are approximated with these *primitive geometries*, **tessellate**

Real-time graphics – ideas

- Let's not calculate unnecessarily: as soon as possible, filter out the geometries that definitely won't be displayed on the screen
- In addition, perform all operations in a coordinate system in which it is easiest to calculate
- And we reuse the results of our previous calculations wherever we can

Incremental image synthesis – terms

- *Coherence*: Instead of pixels, we start with larger logical units, *primitives*
- Accuracy: *object space accuracy* (instead of "pixel accuracy")
- *Clipping*: remove elements sticking out of the screen (out of sight), do not calculate with them unnecessarily
- *Incremental principle*: For the shading and the occlusion tasks, we use the information obtained from the larger unit (for example, the slope of the triangles $(\partial_x z, \partial_y z)$ to calculate the depth of the fragments).

Comparison

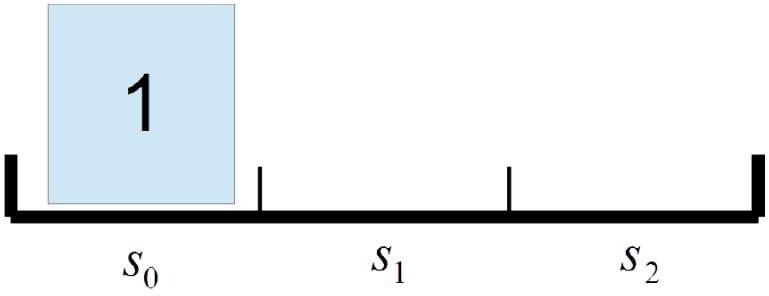
Ray tracing

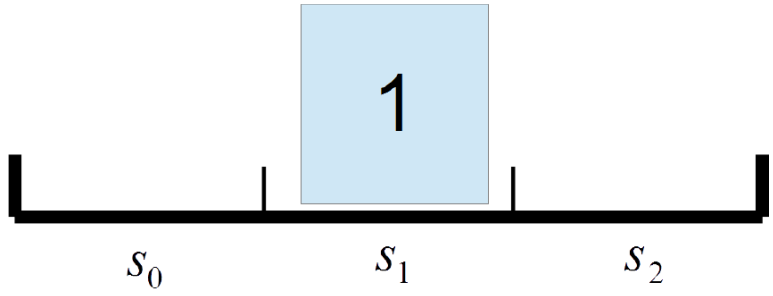
- per pixel calculation
- if it can be intersected with a ray, it can be used
- we get reflection, refraction, shadows
- occlusion task is trivial
- many pixels, many rays, high computational demand

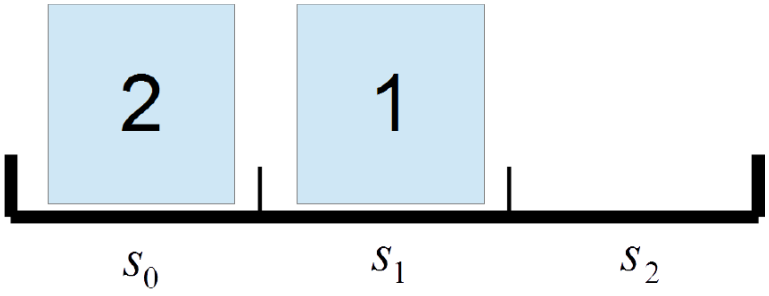
Incremental rendering

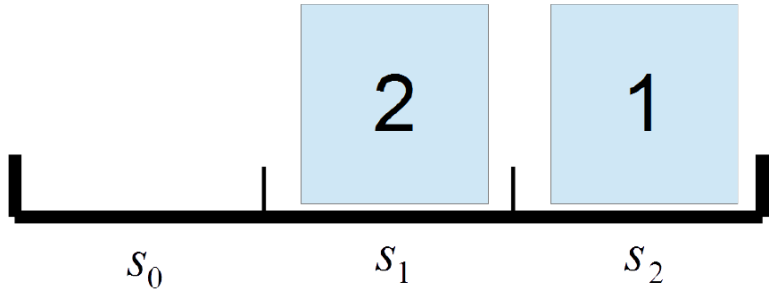
- per primitive calculation
- if not a primitive, it must be approximated with primitives
- a separate algorithm is required for these
- must be solved separately
- lower computational demand due to coherence

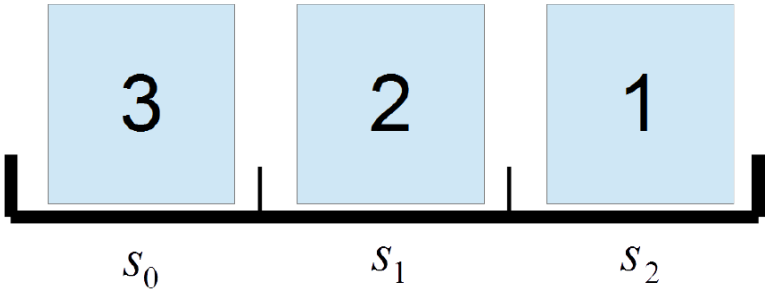
Pipeline











Pipeline

- The pipeline is a chain of processing units
- The input of s_i processing unit is the output of s_{i-1} processing unit
- The output of s_i is the input of s_{i+1}
- If we can divide a problem into n consecutive sub-tasks, and these sub-tasks can be completed in roughly the same time, then we can work on n sub-task simultaneously in a unit of time
- Except for the initial run-up and the final "run-down" starting with the placement of the last workpiece

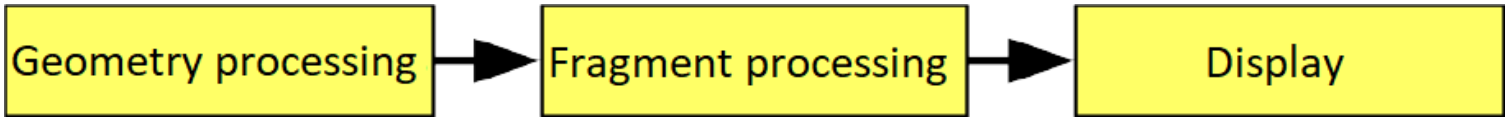
Graphics pipeline

- The sequence of operations for creating an image of our scene is called **graphics pipeline**
- In essence, real-time applications only have to provide a description of our scene, the steps of image synthesis are performed by the graphics pipeline
- In the pipeline, there are several coordinate system changes - we try to perform every task in a coordinate system that best fits it

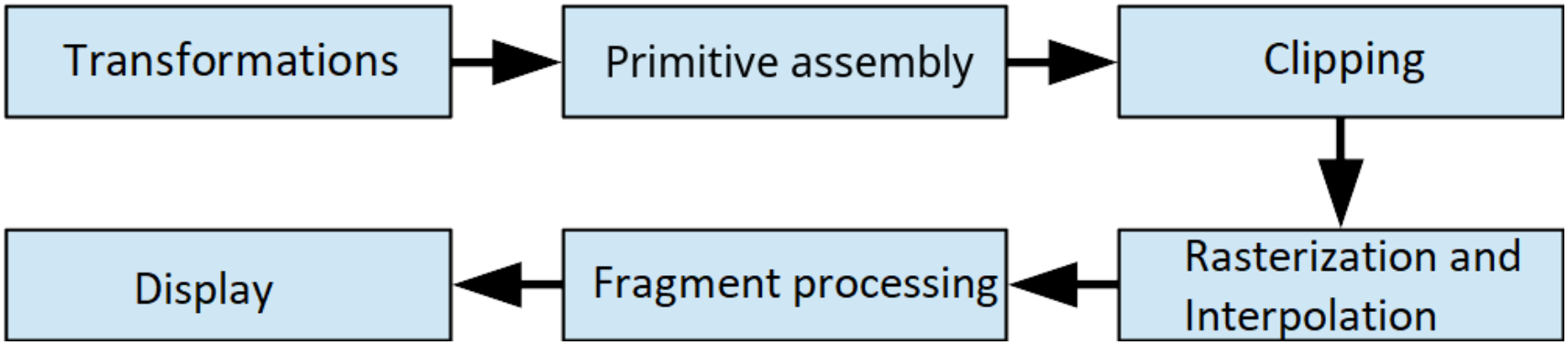
Operation of pipeline

- It starts after every primitive drawing command (e.g glDrawArrays)
- The primitives to be drawn go through all the steps of the pipeline independently (usually parallel to each other).
- It can be described and grouped in several ways

Graphics pipeline



Graphics pipeline



Graphics pipeline

- Its steps in terms of main operations:
  - Transformations
  - Primitive assembly
  - Clipping
  - Homogeneous division
  - Rasterisation and interpolation
  - Fragment processing
  - Displaying (and handling occlusion)
- The result of the graphics pipeline is an image (a two-dimensional array of pixels, each element containing a color value)

Input data of the pipeline

- *Geometric* and *optical model* of the objects to be drawn
- Properties of the *virtual camera* (point of view, viewing angle)
- Canvas (block of pixels on which we map the plane projection of our scene)
- *Lighting data* for light sources and materials in the scene

Transformations

- The task of the transformations in the pipeline: to "deliver" the given object in the model space to the screen space
- Steps (summary):

model CS
→ world CS
→ camera CS
→ normalized device CS
→ screen CS

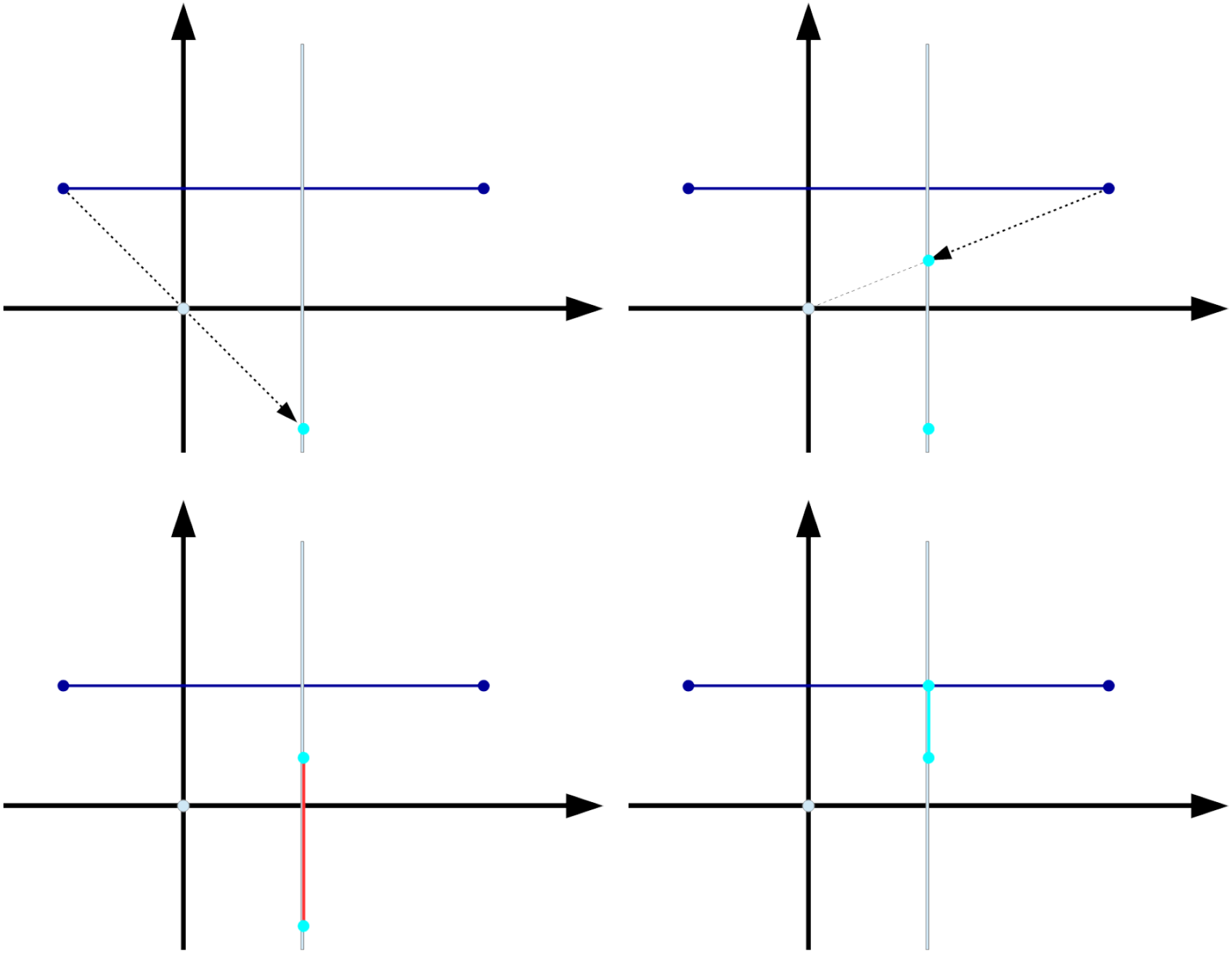
Transformations

- For primitives it is sufficient (*) to transform their vertices and then connect the transformed vertices
- Therefore, the transformations are performed on the vertices of the primitives
- (*): This should be handled with care for central projection!

Transformations

- Before clipping, we do not make a homogeneous division
- So, the phase called Transformations does not go all the way to the screen C.S, it stops in the homogeneous space after central projection, **before** the homogenous division.
- In practice this is what we set in the vertex shader the gl_Position variable.

Segment reversal problem



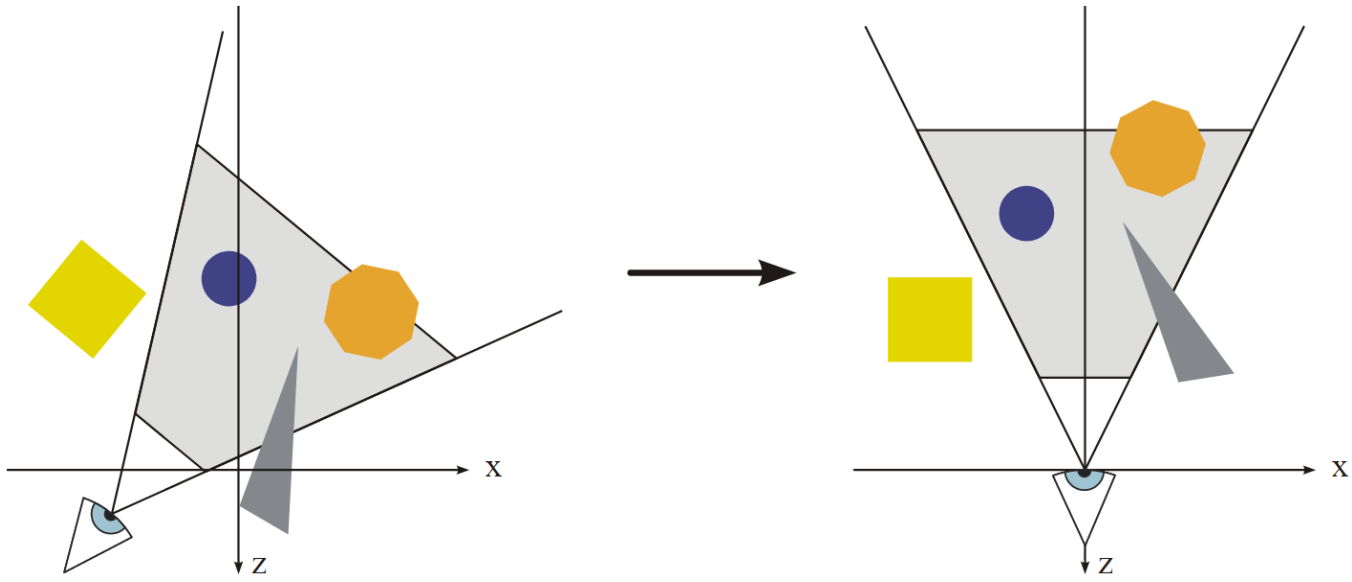
Coordinate systems

- Normalized device CS:
  Specific to the hardware / API, $[-1,1] \times [-1,1] \times [-1,1]$ or $[-1,1] \times [-1,1] \times [0,1]$ dimensional CS.
- Screen CS:
  CS corresponding to the image to be displayed on the screen/window (left-handed, upper-left "corner" is the origin).

Model (world) transformation

- It places the models given in its own (model) coordinate system into the world coordinate system
- It is typically different for each model (maybe the two elements of our scene only differ in the world transformations!)
- Mostly affine transformations
- In practice: this is the *model* (or *world*) matrix in our code

View (camera) transformation



- It transforms the world coordinate system into a coordinate system fixed to the camera
- We get the transformation from the properties of the camera.
- In practice: this is the *view* or *camera* matrix.

View (camera) transformation

- Properties are the same as for ray tracing:
  $\mathbf{eye}, \mathbf{center}, \mathbf{up}$
- From this we get the axes of the view coordinate system:

$$\mathbf{w} = \frac{\mathbf{eye} - \mathbf{center}}{|\mathbf{eye} - \mathbf{center}|}$$

$$\mathbf{u} = \frac{\mathbf{up} \times \mathbf{w}}{|\mathbf{up} \times \mathbf{w}|}$$

$$\mathbf{v} = \mathbf{w} \times \mathbf{u}$$

View (camera) transformation

- Converting into a coordinate system with **eye** origin and $\mathbf{u}, \mathbf{v}, \mathbf{w}$ axes:

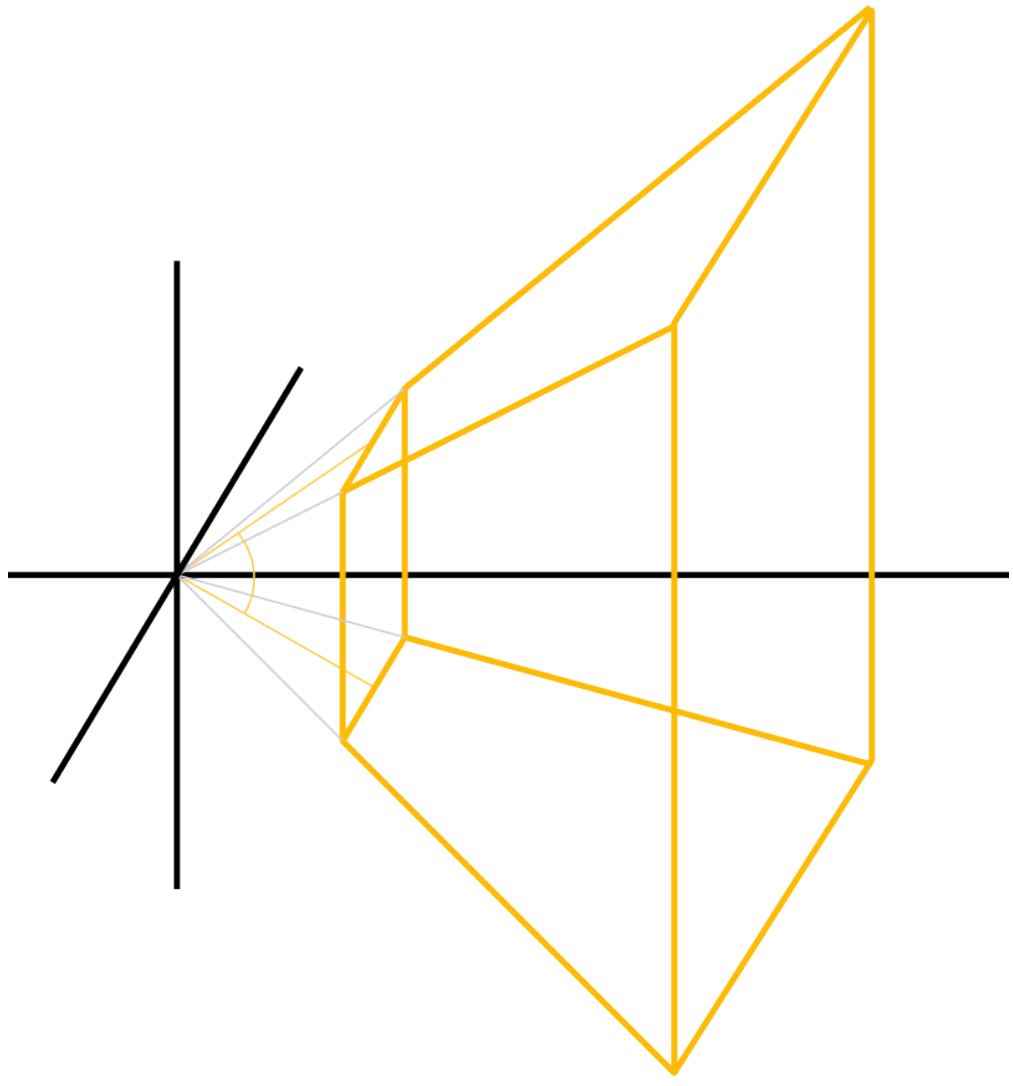
$$T_{View} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Projection – parallel projection

- For example the matrix for projecting onto the XY plane

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Projection – central projection



Projection – central projection

- Reminder: 3. lecture
- It transforms the space inside the viewing frustum into normalized CS
- What was in the frustum, it will be in $[-1,1] \times [-1,1] \times [0,1]$ (or $[-1,1] \times [-1,1] \times [-1,1]$) range
- The transformation makes parallels from the *projection lines* passing through the camera
- The transformation pushes the camera position to infinity
- In practice: this is the *proj* matrix

Projective transformation

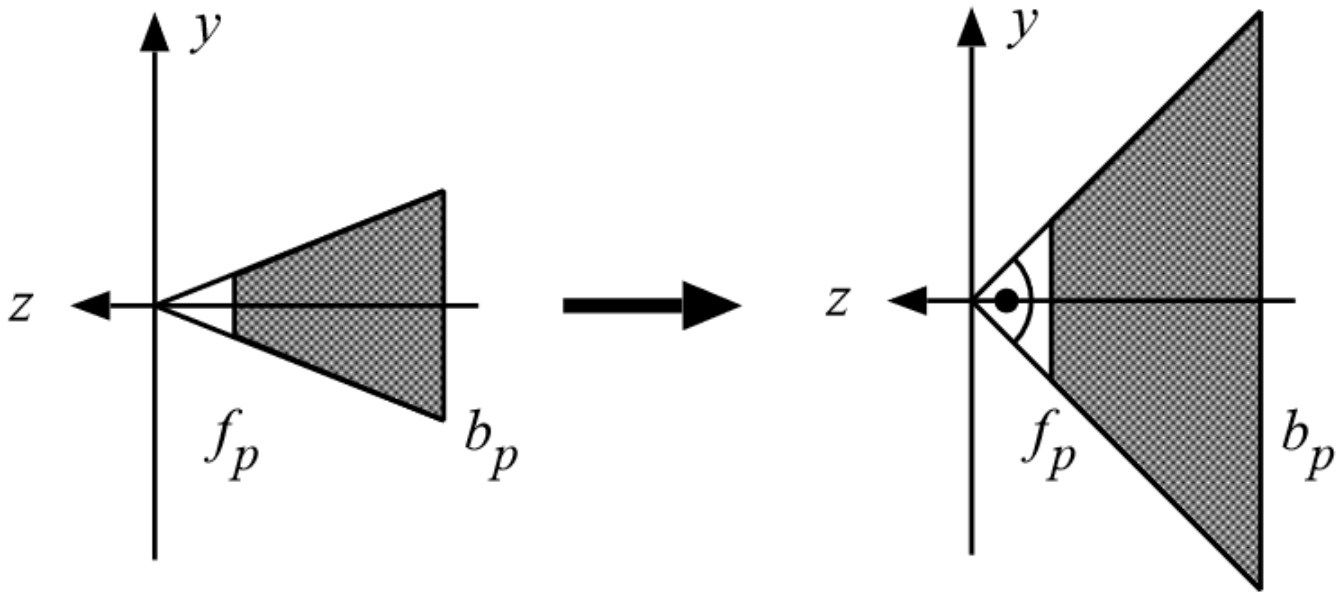
- Reminder: properties
  - vertical and horizontal opening angle *(fovx, fovy)* or the ratio of the sides of the base and the vertical opening angle *(fovy, aspect)*,
  - distance of the near clipping plane *(near)*,
  - distance of the far clipping plane *(far)*

Projective transformations in the pipeline

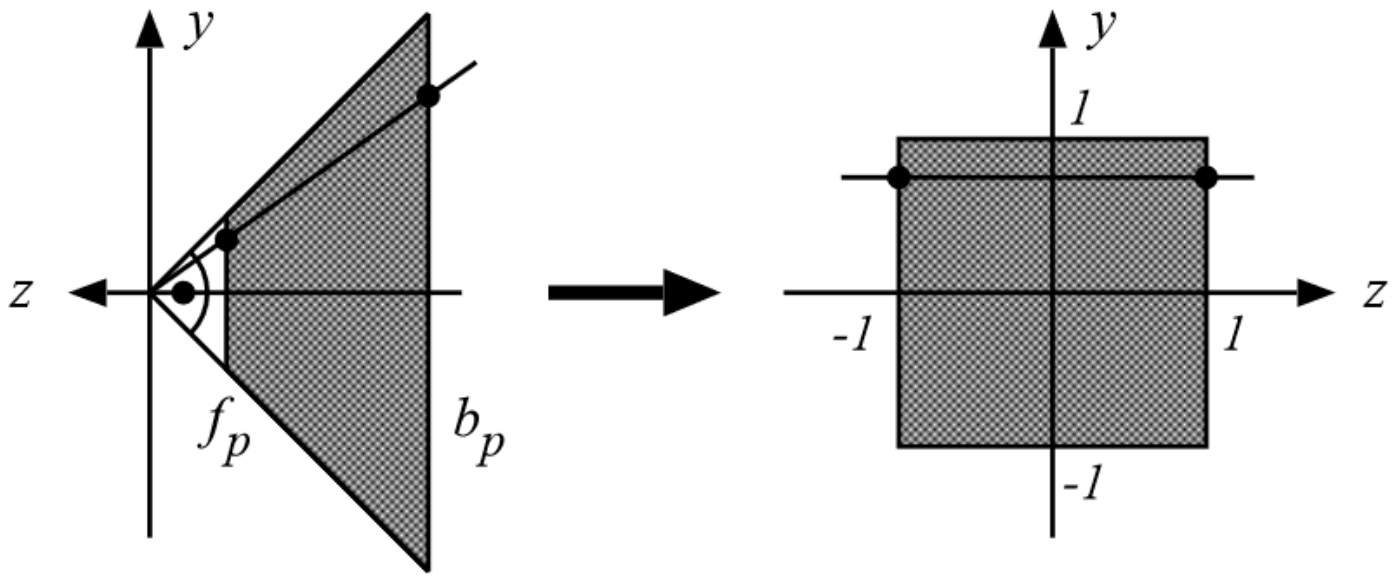
1. Let's normalize the viewing frustum so that both angles are $\frac{\pi}{2}$
2. Let's perform the central projection (from the origin, to a plane 1 unit away from the origin, parallel to the XY plane)
3. Transform the $z = \mathit{near}$, $z = \mathit{far}$ planes into the $z = -1$ and $z = 1$ planes, respectively

Thus, with (1)–(2) we normalize the result's x and y coordinates (map them into $[-1, 1]$), and with (3) we map the z component

Normalizing the viewing frustum (1)



Normalizing the viewing frustum (2–3)



Normalizing the viewing frustum (1)

- Pay attention: at this point of our pipeline, the camera faces $-Z$ and it is at the origin
- From the space above, let's move to a more "normalized" frustum – whose opening angle along x and y is 90 degree
- Matrix form:

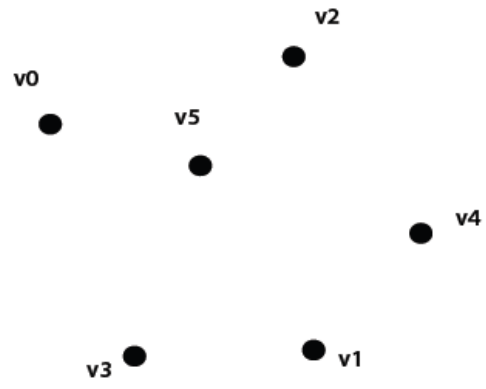
$$\begin{bmatrix} 1/\tan\frac{fovx}{2} & 0 & 0 & 0 \\ 0 & 1/\tan\frac{fovy}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

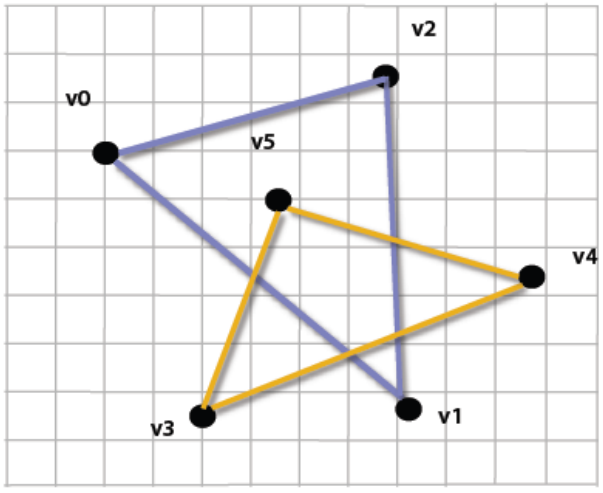
Normalizing the depth (2–3)

- After that, only the z coordinates of the near and far clipping plane needs to be mapped according to the normalization (to $-1, 1$ or $0, 1$):

$$T_{Projection} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{near+far}{near-far} & \frac{2 \cdot near \cdot far}{near-far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

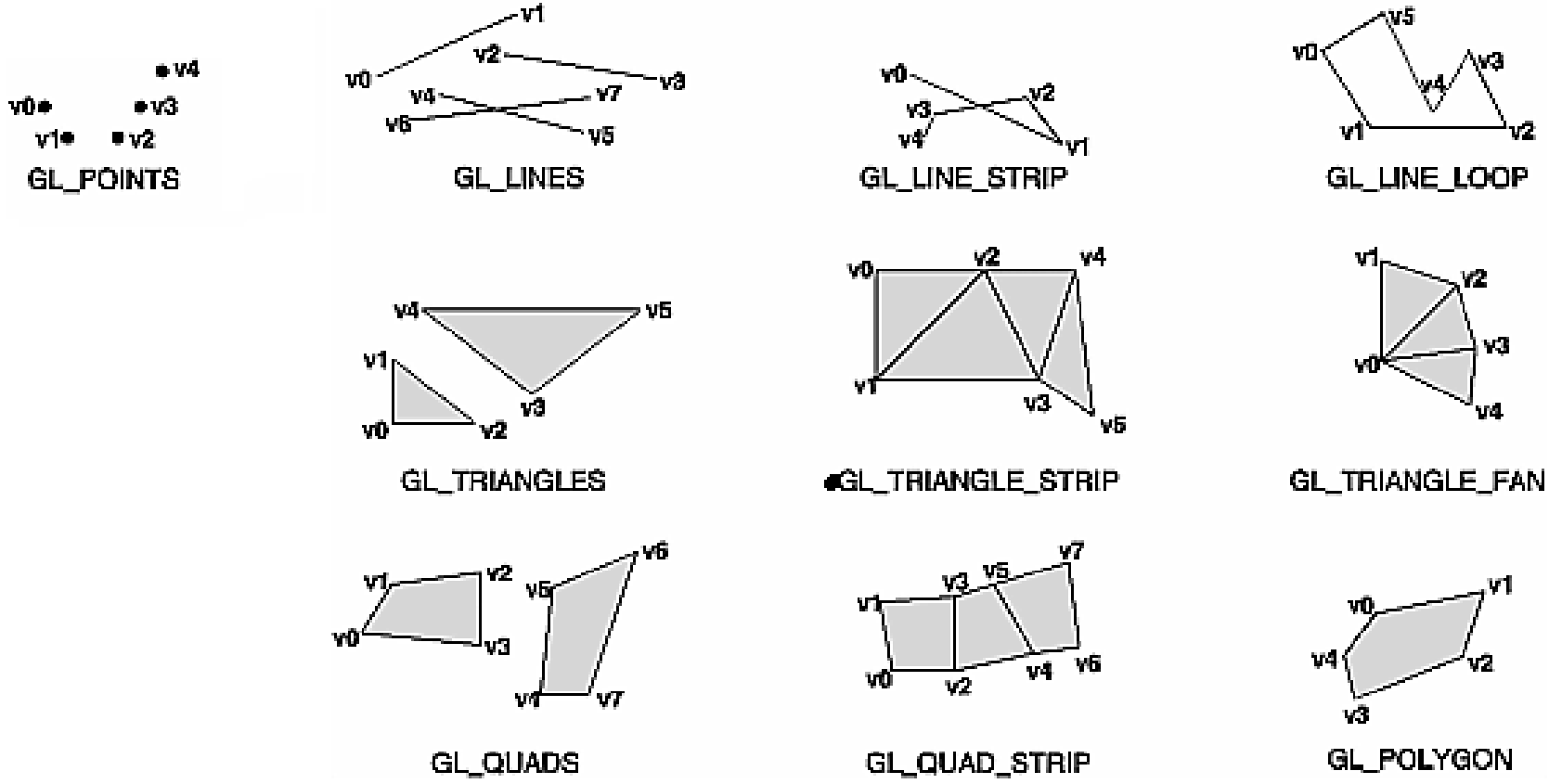
Primitive assembly





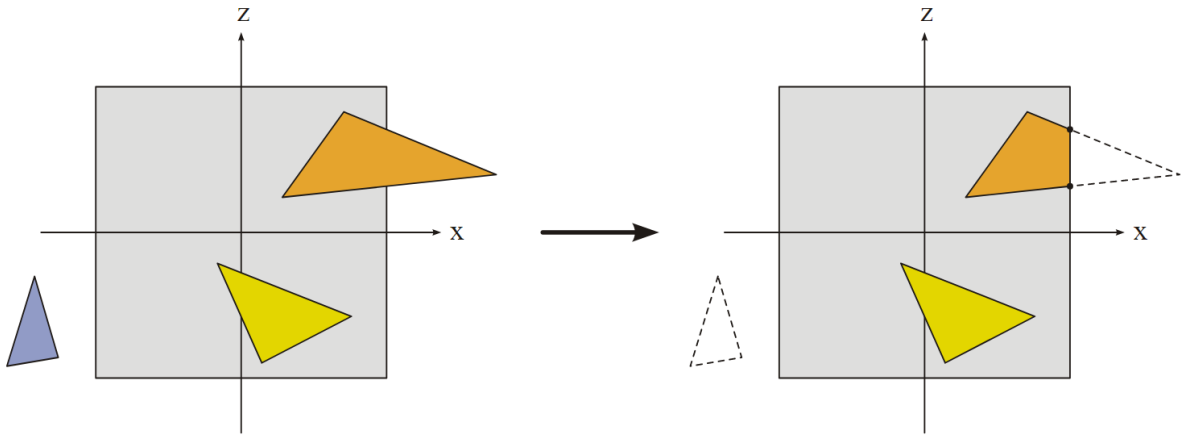
- The vertices placed in the normalized device CS are connected as primitives
- We distinguish three types of primitives: points, segments, and triangles
- Further variants based on the strategy for connecting vertices

Primitives



In modern OpenGL GL_QUADS, GL_QUAD_STRIP, GL_POLYGON are deprecated.

Clipping



- Goal: don't work unnecessarily with elements that won't be pixels (they will not appear)
- To do this, we filter out those elements that will *definitely* not be displayed on the screen
- And we also *clip* those that are only partially on the screen (identify the pieces of it, that are completely inside the screen and cover these pieces with primitives)

Clipping in homogeneous coordinates

- Now we only consider clipping a single point, in homogeneous coordinate system:
- Let: $[x_h, y_h, z_h, h]^T = \mathbf{M}_{\text{Proj}} \cdot [x_c, y_c, z_c, 1]^T$
- Goal:
  $[x, y, z]^T := [x_h/h, y_h/h, z_h/h]^T \in [-1, 1] \times [-1, 1] \times [-1, 1]$,
  i.e

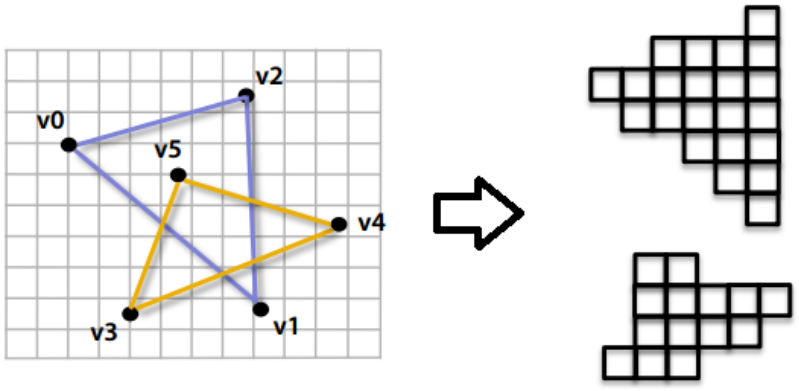
Let $h > 0$, and

$$-1 \leq x \leq 1$$
$$-1 \leq y \leq 1$$
$$-1 \leq z \leq 1$$

From this we get:

$$-h \leq x_h \leq h$$
$$-h \leq y_h \leq h$$
$$-h \leq z_h \leq h$$

Rasterization

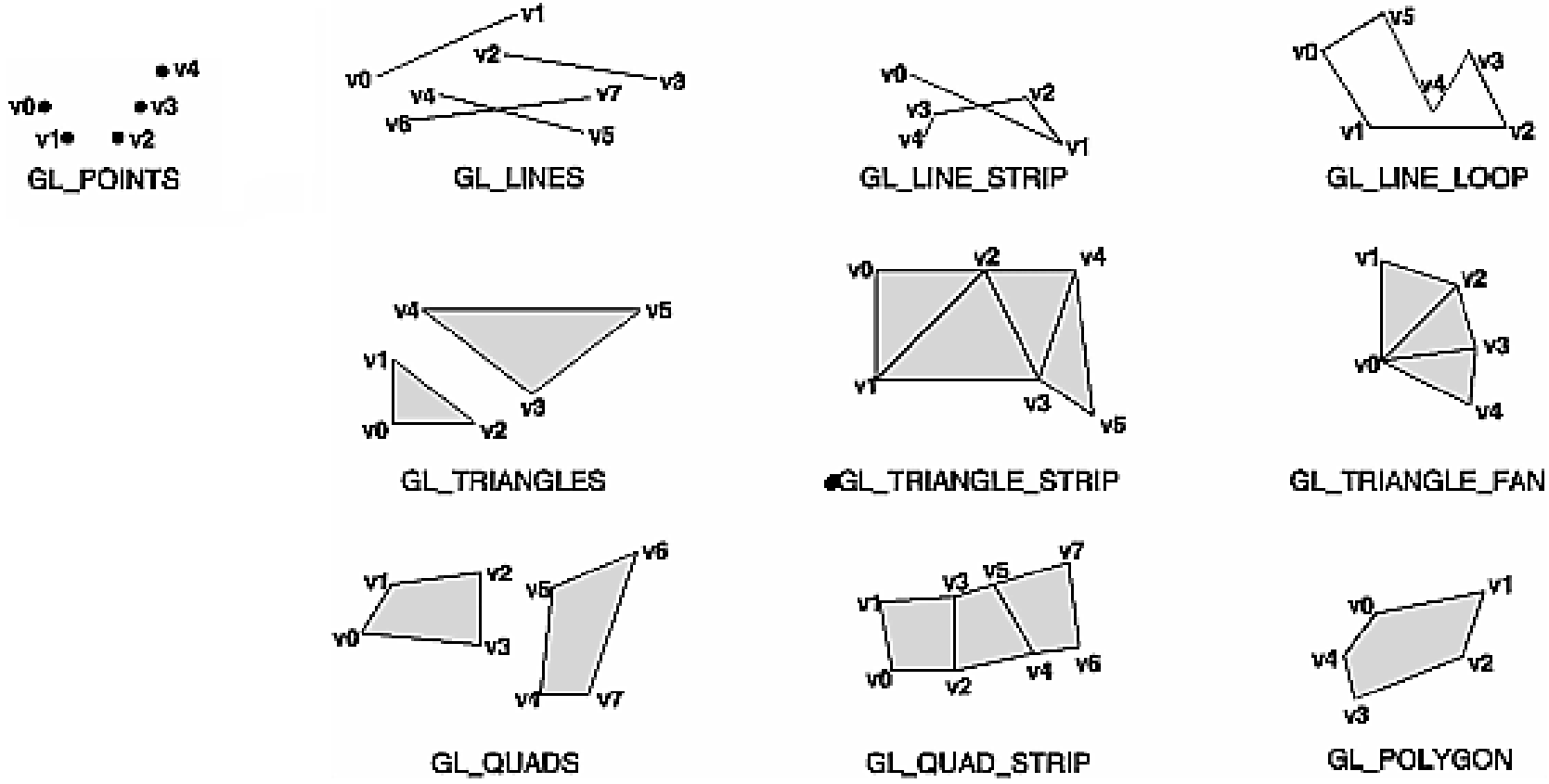


- Remember: so far, all the primitives we have talked about were continuous
- However, we have to work in a discrete space, on the pixels of the screen
- In other words, we have to discretize our continuous primitives – this is called *rasterization*

Rasterization – Why triangles?

- We need to choose geometric primitives that we can quickly rasterize
- What would be a good primitive? It would be good, e.g. if we could easily calculate from the surface coordinates of a pixel, its neighbouring pixels' coordinates, and if the primitive were in one plane...
- The triangle is like that!
- All other surfaces are approximated with such primitives (in essence: with flat faces). $\leftarrow$ *tessellation* – before using the pipeline

Rasterization – primitives



In modern OpenGL GL_QUADS, GL_QUAD_STRIP, GL_POLYGON are deprecated.

Occlusion task

- Task: to decide what piece of the surface is visible in certain parts of the image.
- Object space algorithms:
  - We work with logical units, it does not depend on the screen resolution.
  - Bad news: it will generally not work.
- Screen space algorithms:
  - We decide what is visible pixel by pixel.
  - Just like with ray tracing.

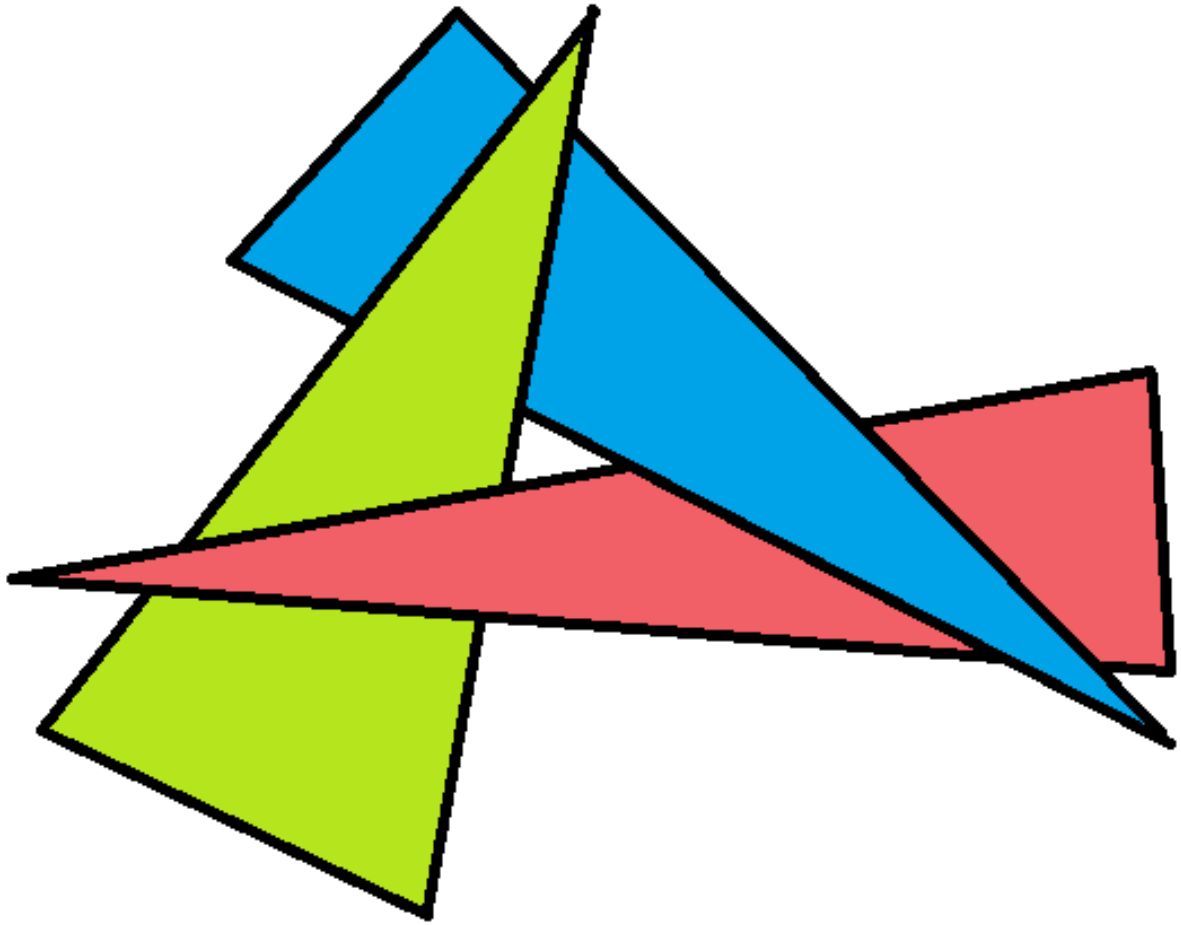
Back-face culling

- Assumption: our objects are "closed", i.e. if we are not inside the object, we cannot see its surface from the inside.
- Traversal order: set the order in which the vertices of the polygons *must* be given:
  - clockwise (CW)
  - counter clockwise (CCW)
- If, after the transformations, the order of the vertices is not the same as specified, we see the face *from the back* $\Rightarrow$ no need to draw it, *discardable*.

Painter's algorithm

- We draw the polygons in order of back to front!
- What is closer, we will draw later $\Rightarrow$ whatever is further away will be covered.
- Problem: how do we order the polygons?
- Even with few triangles, there are cases where it is not possible to give a clear order.

Painter's algorithm



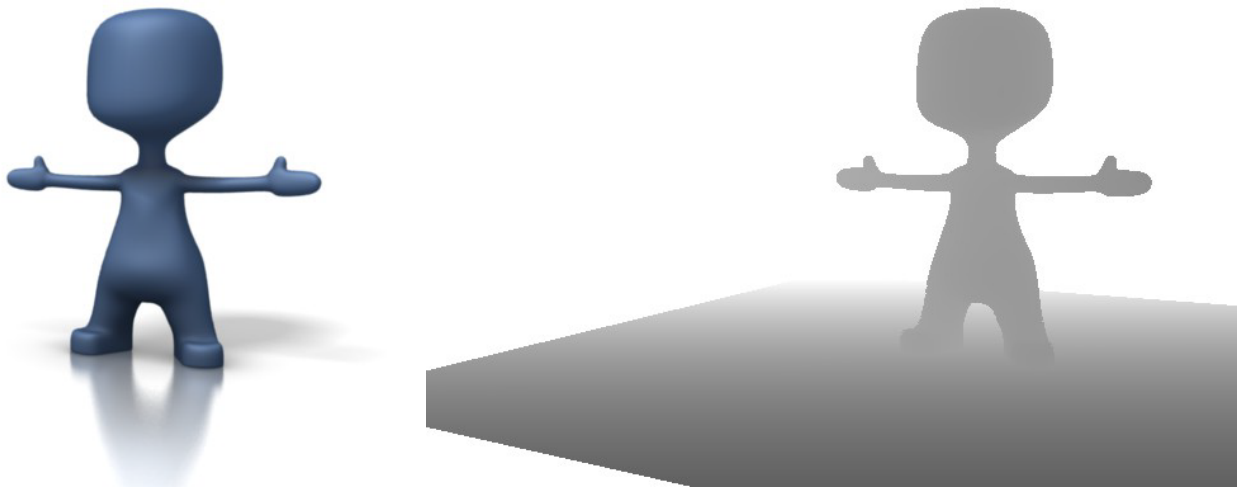
Z-buffer algorithm

- Screen space algorithm
- For each pixel, we store the corresponding depth value.
- If we were to draw on this pixel again (*Z-test*):
  - **If** the new Z value is "deeper", **then** the point is occluded $\Rightarrow$ we don't draw
  - **If** the old Z value is "deeper", **then** the new point occludes it $\Rightarrow$ we draw *and* store the new Z value.

Z-buffer

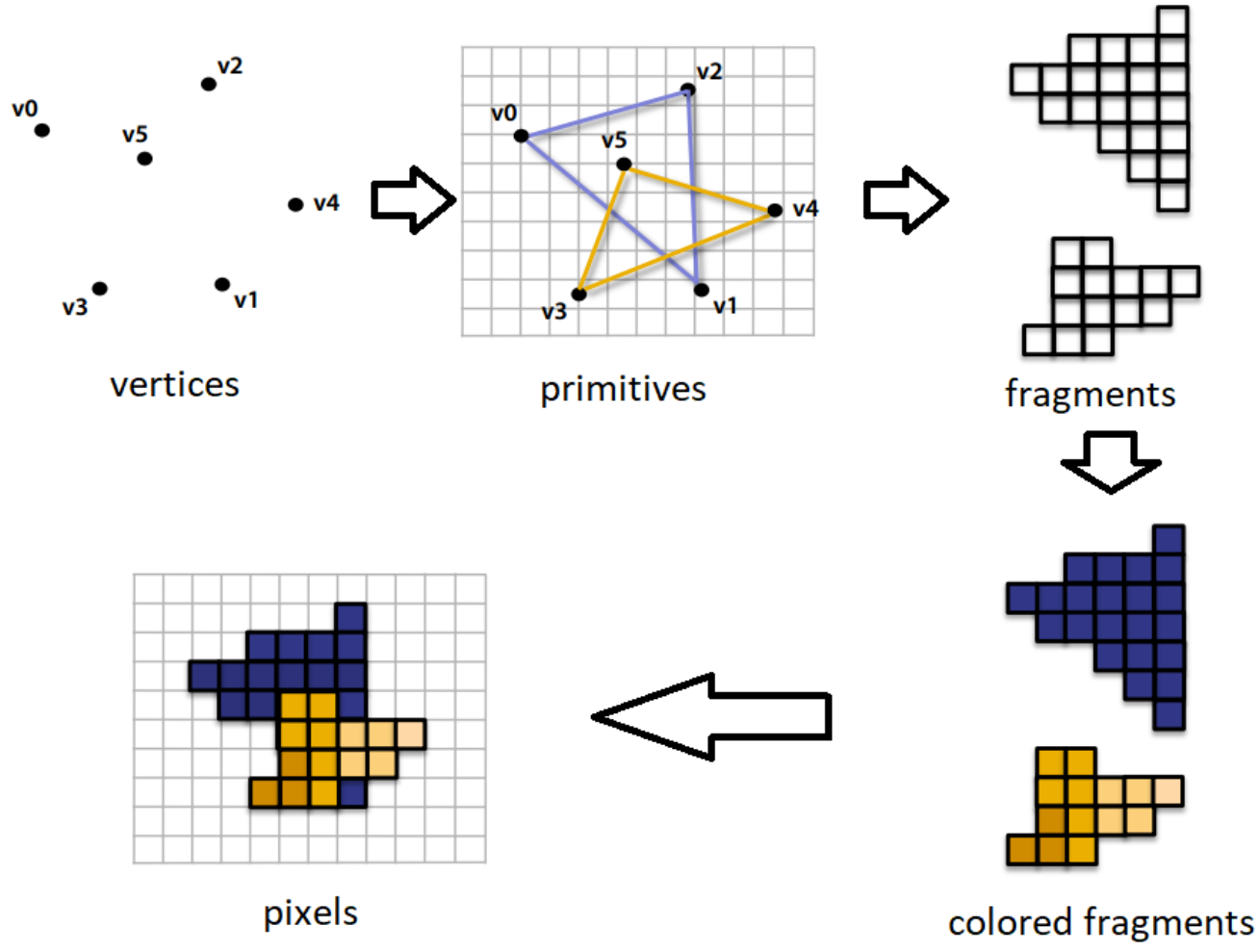
- Z-buffer or *depth buffer*: separate memory area.
- An array with the same size as the screen/window.
- Accuracy: depends on the distance between the near and far clipping plane.
- Each pixel has a depth value in the buffer.
- We need to compare with this and write here if the pixel *passed the Z-test*.
- In practice:
  - 16-32 bit element size
  - Hardware acceleration
  - E.g. near clipping plane: t, distant: $1000t$, then 98% of the Z-buffer describes the first 2% of the range.

Z-buffer



by macouno, macouno.com

GPU pipeline



Local illumination

- Once we have the mappings of our primitives to pixels, let's somehow calculate colors

Shading with a single color

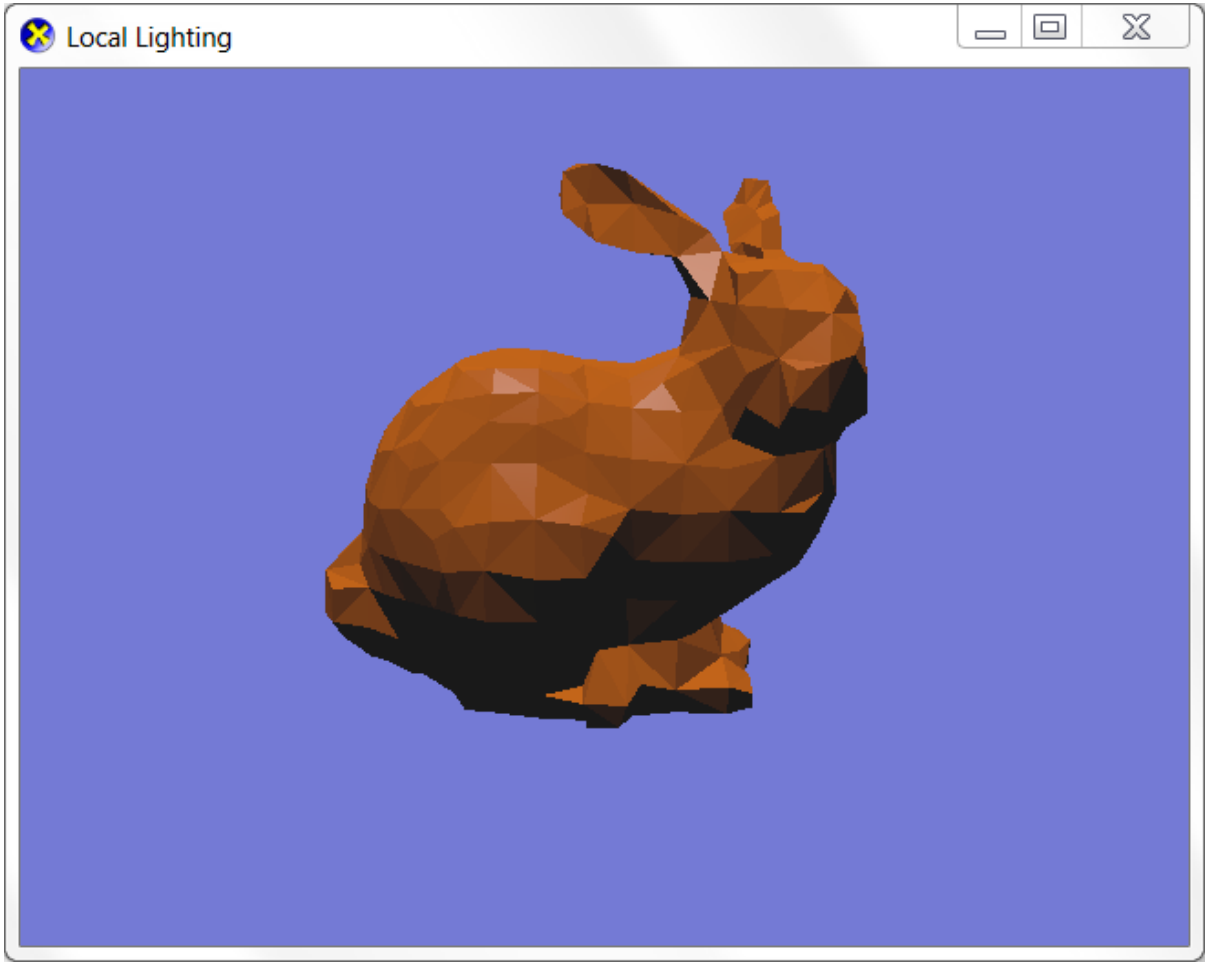
- We assign a color to each object/primitive, and this will be the value of the pixels when drawn.
- Fastest: illumination is practically only one value assignment.
- Horrible: neither realistic nor asthetic.

Shading with a single color

Flat shading

- The lighting is calculated once per polygon, the color is homogeneous within the face.
- Fast: the number of operations depends on the number of polygons, independent of the number of pixels.
- It can be used: for diffuse, solid-colored objects without curved parts.

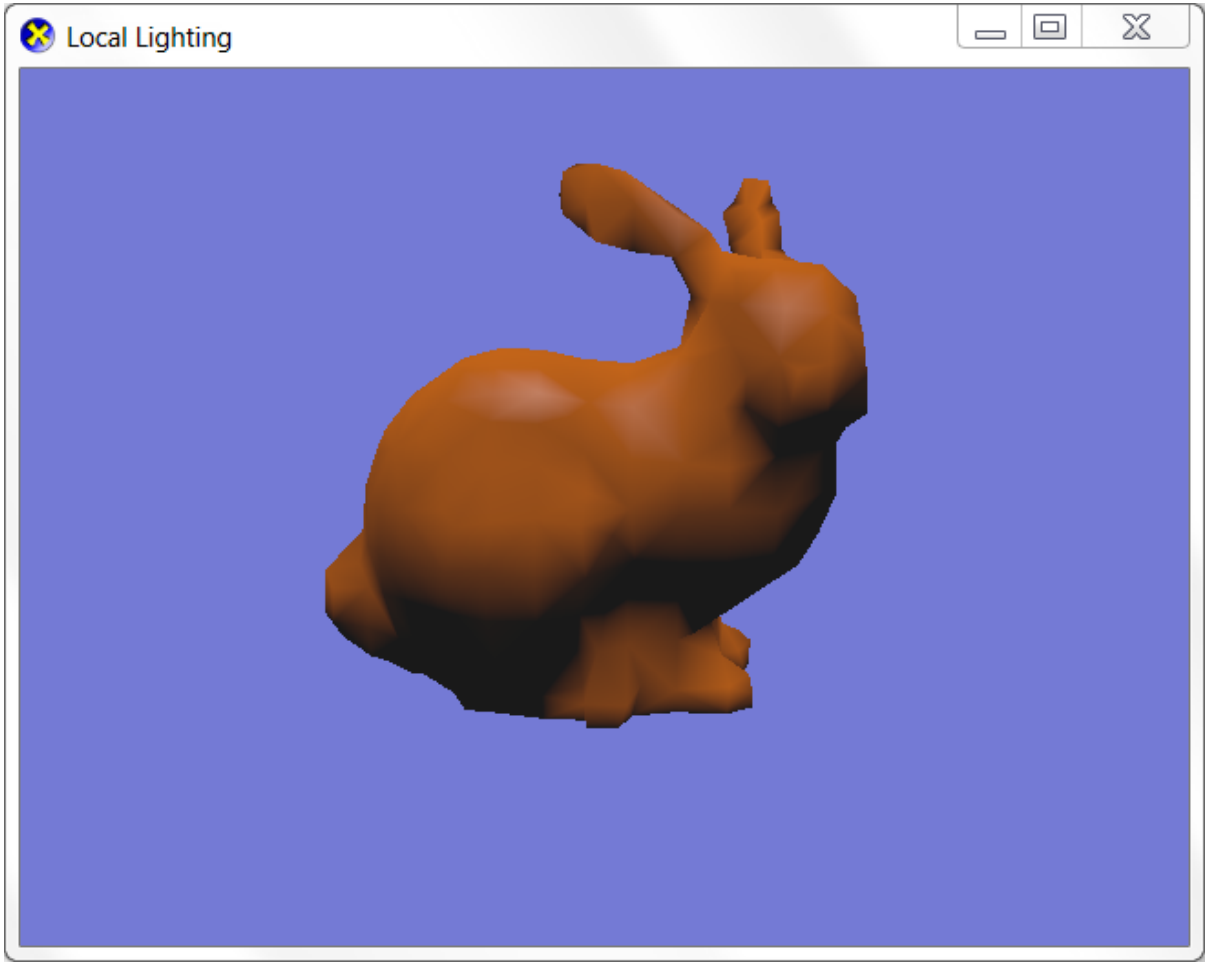
Flat shading



Gouraud shading

- The lighting is calculated per vertex, and we get the colors by interpolating the values from the three vertex.
- Slower: N lighting calculations + interpolation for each pixel.
- Better: the quality of the shading largely depends on the number of polygons. But specular highlights cannot appear on large faces.
- It is well suited for the graphics pipeline: in addition to calculating the position of the vertices, we can also calculate the color, after which the resulting fragments will automatically receive the interpolated colors during rasterization.

Gouraud shading



Phong shading

- Only the normal vectors are interpolated, the illumination is calculated per pixel.
- Slowest: must be calculated at *each pixel*.
- Best of all: the quality of shading does not depend on the number of polygons. Specular highlights can even appear in the center of the polygon.
- It is well suited for graphics pipeline: during rasterization, the normal vectors are interpolated and the color is calculated separately for each fragment using the obtained normal vector.

Phong shading