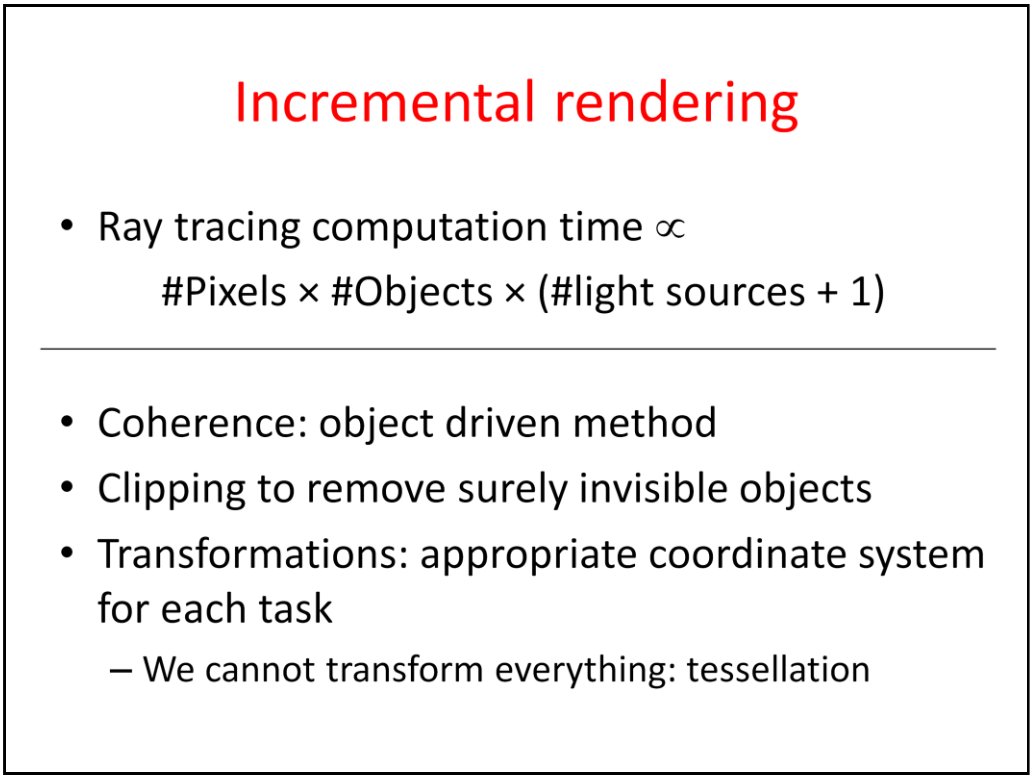
Incremental 3D rendering

Szirmay-Kalos László

This lecture reviews the 3D rendering pipeline implemented by current GPUs.



Incremental rendering

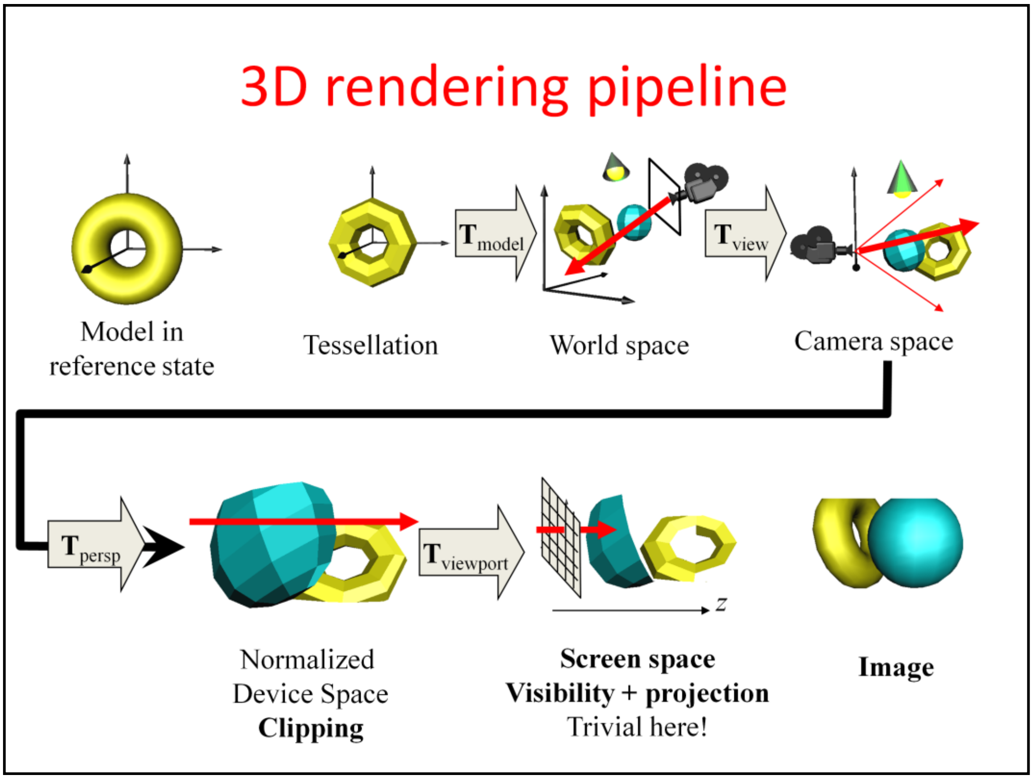
- Ray tracing computation time $\propto$

  #Pixels × #Objects × (#light sources + 1)

- Coherence: object driven method
- Clipping to remove surely invisible objects
- Transformations: appropriate coordinate system for each task
  - We cannot transform everything: tessellation

Ray tracing processes each pixel independently, thus it may repeat calculations that could be reused in other pixels. Consequently, ray tracing is slow, it is difficult to render complex, animated scenes with ray tracing at high frame rates.

The goal of incremental rendering is speed and it sacrifices everything for it. To obtain an image quickly, it solves many problems for regions larger than for a single pixel. This region is usually a triangle of the scene. It gets rid of objects that are surely not visible via clipping. Most importantly, it executes operations in appropriate coordinate systems where this particular operation is simple (recall that ray tracing does everything in world space). Moving object from one coordinate system to another requires transformations. As we cannot transform all types of geometry without modifying the type, we approximate all kinds of geometry with triangles. This process is called tessellation.



3D rendering pipeline

Model in reference state

Tessellation

T_{model}

World space

T_{view}

Camera space

T_{persp}

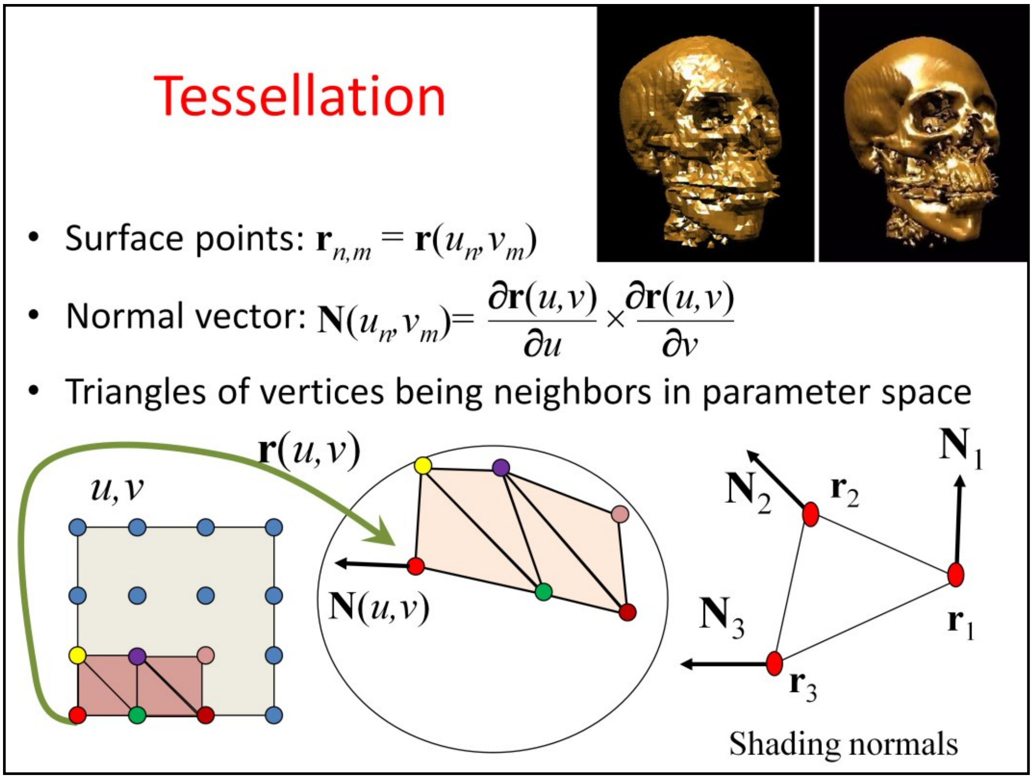
Normalized Device Space **Clipping**

$T_{viewport}$

Screen space Visibility + projection Trivial here!

Image

Incremental rendering is a pipeline of operations starting at the model and ending on the image. Objects are defined in their reference state. First surfaces are approximated by triangle meshes. Then the tessellated object is placed in the world, setting its real size, orientation and position. Here, different objects, the camera and light sources meet. Ray tracing would solve the visibility problem here, but incremental rendering applies transformations to find a coordinate system when it is trivial to decide whether two points occlude each other, and projection is also simple. This is the screen coordinate system where rays are parallel with axis z and a ray has the pixel's x,y coordinates. To transform the model from world to screen coordinates, first we execute the camera transformation which translates and rotates the scene so that the camera is in the origin and looks at the –z direction (the negative sign is due to the fact that we prefer right handed coordinate system here). In the camera coordinate system, projection rays go through the origin and projection is perspective. To simplify this, we distort the space and make rays meet in an ideal point at the end of axis z, so projection rays will be parallel. Clipping is executed here. Finally, we take into account the real resolution of the image and scale the space accordingly.



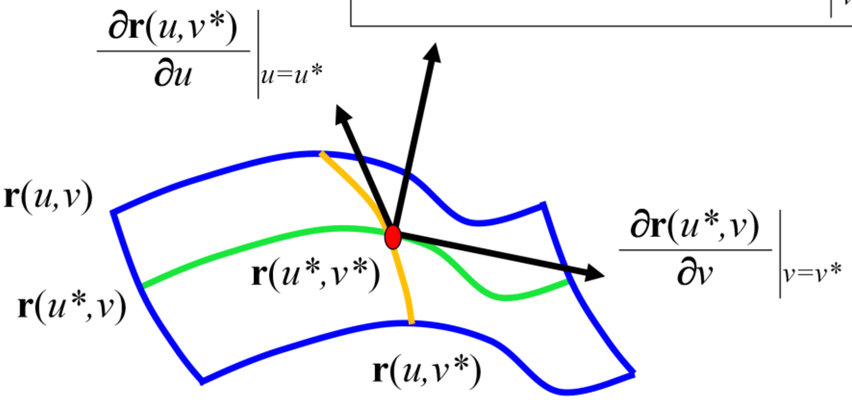
Tessellation

- Surface points: $\mathbf{r}_{n,m} = \mathbf{r}(u_n, v_m)$
- Normal vector: $\mathbf{N}(u_n, v_m) = \frac{\partial \mathbf{r}(u,v)}{\partial u} \times \frac{\partial \mathbf{r}(u,v)}{\partial v}$
- Triangles of vertices being neighbors in parameter space

The tessellation of a parametric surface basically means the evaluation of the surface equation in parameter points that are placed regularly by a grid. Those points form a triangle that are neighbors in parameter space. To obtain the shading normals, the cross product of the partial derivatives of the parametric equation is also computed at the sample points. The triangles of the parameter space can be visited row by row. Vertices of a single row are conveniently organized into a triangle mesh.

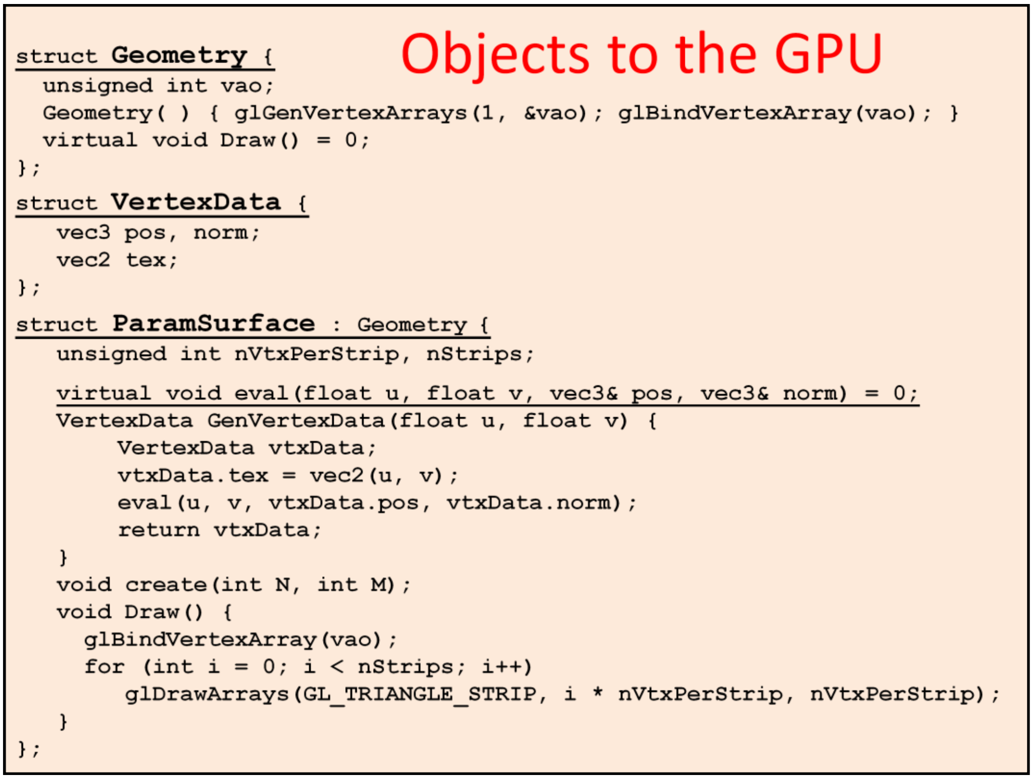
Normal vector of parametric surfaces

$$\mathbf{N}(u^*,v^*) = \left.\frac{\partial \mathbf{r}(u,v)}{\partial u} \times \frac{\partial \mathbf{r}(u,v)}{\partial v}\right|_{\substack{u=u^* \\ v=v^*}}$$


$\left.\frac{\partial \mathbf{r}(u,v^*)}{\partial u}\right|_{u=u^*}$
$\mathbf{r}(u,v)$
$\mathbf{r}(u^*,v)$
$\mathbf{r}(u^*,v^*)$
$\left.\frac{\partial \mathbf{r}(u^*,v)}{\partial v}\right|_{v=v^*}$
$\mathbf{r}(u,v^*)$

To get the normal vector of a parametric surface, we exploit isoparametric lines. Suppose that we need the normal at point associated with parameters u*, v*. Let us keep u* fixed, but allow v to run over its domain. This r(u*,v) is a one-variate parametric function, which is a curve. As it always satisfies the surface equation, this curve is on the surface and when v=v*, this curve passes through the point of interest. We know that the derivative of a curve always tangent to the curve, so the derivative with respect to v at v* will be the tangent of a curve at this point, and consequently will be in the tangent plane.

Similarly, the derivative with respect to u will always be in the tangent plane. The cross product results in a vector that is perpendicular to both operands, so it will be the normal of the tangent plane.



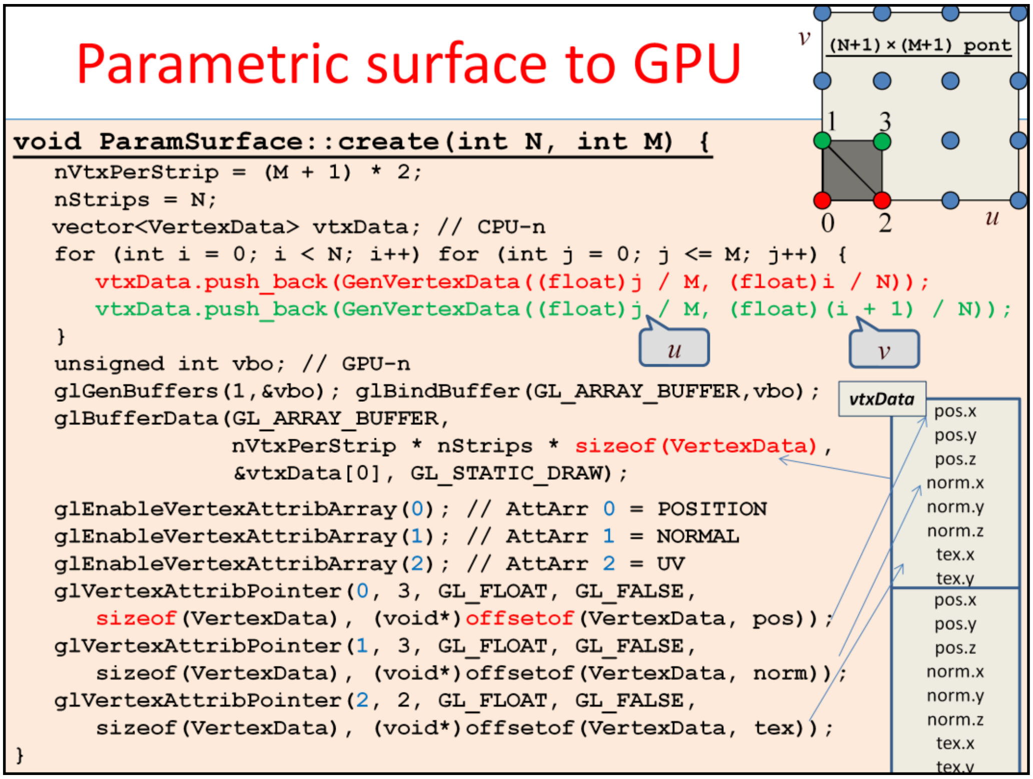
Objects to the GPU

```
struct Geometry {
  unsigned int vao;
  Geometry( ) { glGenVertexArrays(1, &vao); glBindVertexArray(vao); }
  virtual void Draw() = 0;
};
struct VertexData {
   vec3 pos, norm;
   vec2 tex;
};
struct ParamSurface : Geometry {
   unsigned int nVtxPerStrip, nStrips;

   virtual void eval(float u, float v, vec3& pos, vec3& norm) = 0;
   VertexData GenVertexData(float u, float v) {
        VertexData vtxData;
        vtxData.tex = vec2(u, v);
        eval(u, v, vtxData.pos, vtxData.norm);
        return vtxData;
   }
   void create(int N, int M);
   void Draw() {
     glBindVertexArray(vao);
     for (int i = 0; i < nStrips; i++)
        glDrawArrays(GL_TRIANGLE_STRIP, i * nVtxPerStrip, nVtxPerStrip);
   }
};
```

Tessellation is done on the CPU with a C++ program. The tessellated triangle mesh is copied to the GPU and assigned to a **vao** (vertex array object).

The general base class of triangle meshes is the **Geometry** that stores the vao and the number of vertices. In its constructor, the vao is generated and is bound, i.e. is made active. **ParamSurface** is the special geometry of parametric surfaces. All parametric surfaces are similar in how they should be tessellated. The process should decompose the unit square into rows and colums and visit vertices rows by row. A single row forms a **GL_TRIANGLE_STRIP**. Vertices are stored with the position of the vertex, the surface normal of the surface here, and the corresponding parameter pair used as texture coordinates. These data are encapsulated into **VertexData**. When a VertexData object is computed, we need the particular definition of the parametric function, which is not available on this abstract level. So se define a pure virtual **eval** function that should be specified in classes derived from ParamSurface. The creation of the surface data on the GPU happens in the **create** function.

When a ParamSurface is drawn, the vao is bound again since other vaos may become active between the construction of this one and its drawing. Then the vao is drawn stating that its vertices define a triangle strip, and rows are drawn separately.



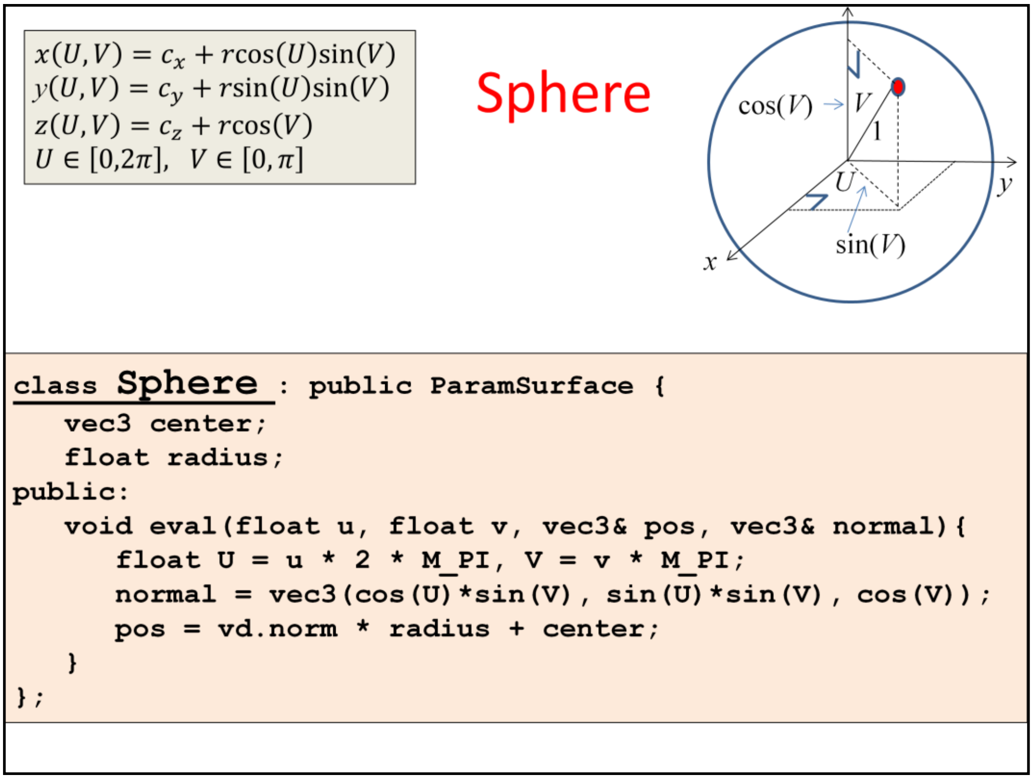
Parametric surface to GPU

```
void ParamSurface::create(int N, int M) {
   nVtxPerStrip = (M + 1) * 2;
   nStrips = N;
   vector<VertexData> vtxData; // CPU-n
   for (int i = 0; i < N; i++) for (int j = 0; j <= M; j++) {
      vtxData.push_back(GenVertexData((float)j / M, (float)i / N));
      vtxData.push_back(GenVertexData((float)j / M, (float)(i + 1) / N));
   }
   unsigned int vbo; // GPU-n
   glGenBuffers(1,&vbo); glBindBuffer(GL_ARRAY_BUFFER,vbo);
   glBufferData(GL_ARRAY_BUFFER,
                nVtxPerStrip * nStrips * sizeof(VertexData),
                &vtxData[0], GL_STATIC_DRAW);
   glEnableVertexAttribArray(0); // AttArr 0 = POSITION
   glEnableVertexAttribArray(1); // AttArr 1 = NORMAL
   glEnableVertexAttribArray(2); // AttArr 2 = UV
   glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
      sizeof(VertexData), (void*)offsetof(VertexData, pos));
   glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
      sizeof(VertexData), (void*)offsetof(VertexData, norm));
   glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE,
      sizeof(VertexData), (void*)offsetof(VertexData, tex));
}
```

Recall that we have activated the vao. Now its corresponding data stored in a single vbo (vertex buffer object) are generated. If the parameter space is decomposed to N columns and M rows, then we have N strips each containing (M + 1) * 2 vertices.

The vbo is generated and bound, then it is filled up by calling the GenVertexData function on the vertices.

In the GPU, vertex shader input register #0 will get the position (3 floats), #1 the normal (3 floats), and #2 the u,v parameter pair (2 floats).



Sphere

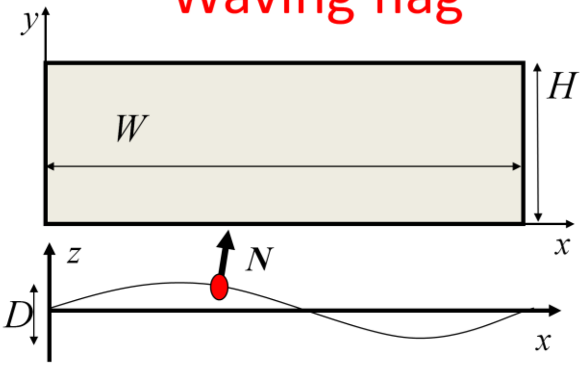
$$x(U,V) = c_x + r\cos(U)\sin(V)$$
$$y(U,V) = c_y + r\sin(U)\sin(V)$$
$$z(U,V) = c_z + r\cos(V)$$
$$U \in [0,2\pi],\quad V \in [0,\pi]$$

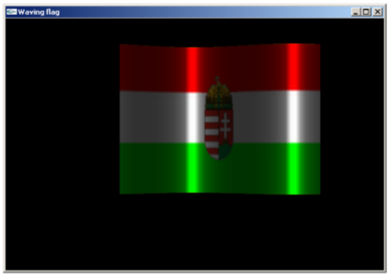
cos(V)
V
1
U
sin(V)
x
y

```
class Sphere : public ParamSurface {
   vec3 center;
   float radius;
public:
   void eval(float u, float v, vec3& pos, vec3& normal){
      float U = u * 2 * M_PI, V = v * M_PI;
      normal = vec3(cos(U)*sin(V), sin(U)*sin(V), cos(V));
      pos = vd.norm * radius + center;
   }
};
```

The construction of a parametric surface is general, the only specific thing is the eval function. So, if we derive a Sphere class from ParamSurface, this virtual function should be implemented according to the equations of the sphere.

Waving flag



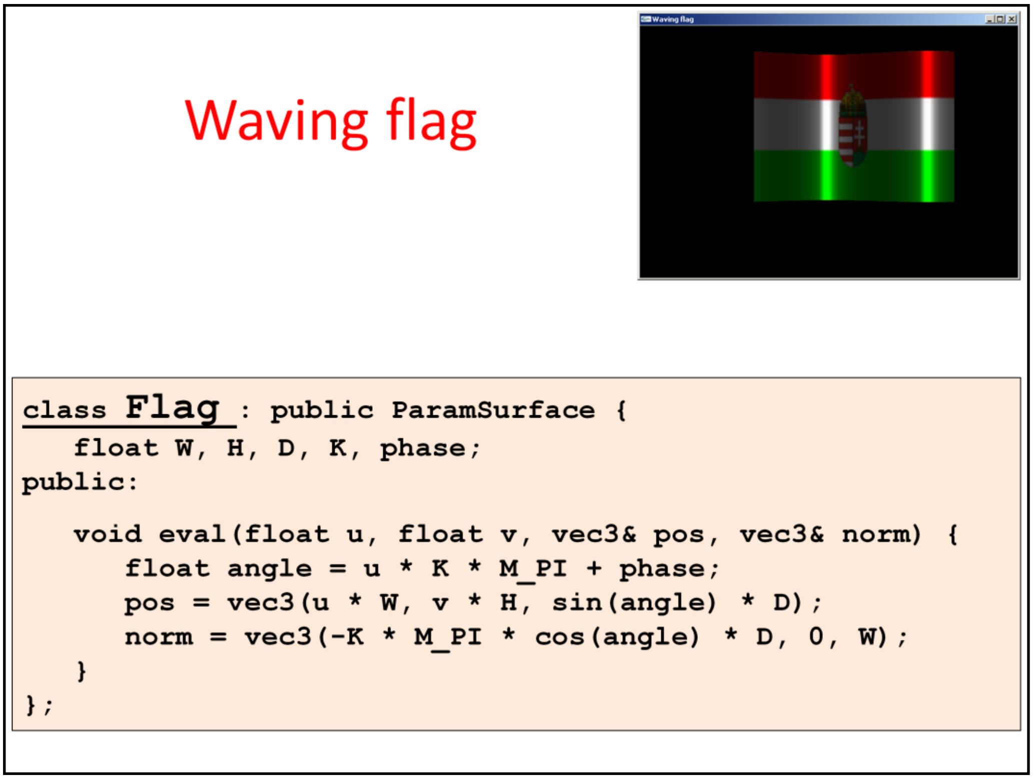


$$\mathbf{r}(u,v) = [u\cdot W,\quad v\cdot H,\quad \sin(K\cdot u\cdot \mathrm{PI}+\mathrm{phase})\cdot D\quad]$$

$$\begin{array}{r} \\ \partial\mathbf{r}/\partial u = \\ \partial\mathbf{r}/\partial v = \end{array}\begin{bmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ W, & 0, & K\cdot \mathrm{PI}\cdot\cos(K\cdot u\cdot \mathrm{PI}+\mathrm{phase})\cdot D \\ 0, & H, & 0 \end{bmatrix}$$

$$\mathbf{n}(u,v)= \partial\mathbf{r}/\partial u \times \partial\mathbf{r}/\partial v =[-K\cdot \mathrm{PI}\cdot \cancel{H}\cdot\cos(K\cdot u\cdot \mathrm{PI}+\mathrm{phase})\cdot D,0,W\cdot \cancel{H}]$$

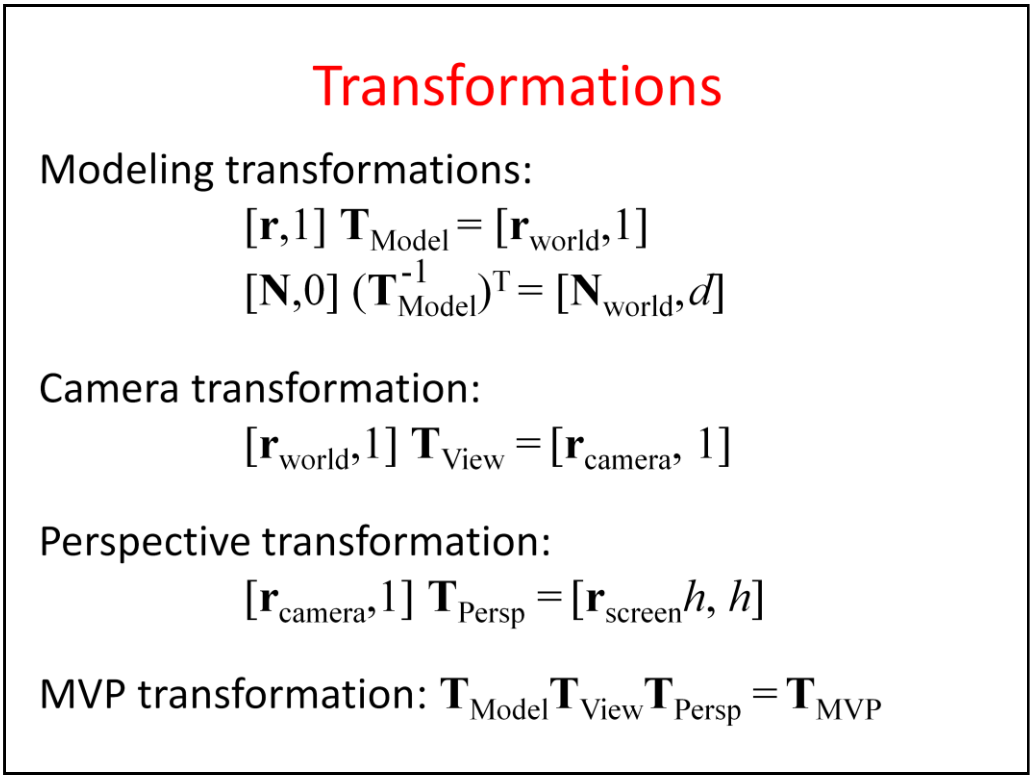
Another example is the waving flag, which is a rectangle that is modulated by a sine wave. As we stated, the normal vector is the cross product of the partial derivatives.



Waving flag

```
class Flag : public ParamSurface {
   float W, H, D, K, phase;
public:
   void eval(float u, float v, vec3& pos, vec3& norm) {
      float angle = u * K * M_PI + phase;
      pos = vec3(u * W, v * H, sin(angle) * D);
      norm = vec3(-K * M_PI * cos(angle) * D, 0, W);
   }
};
```

Again, only the eval function needs to be implemented on this level, which assigns the position and the normal vector to parameter pair u and v.



Transformations

Modeling transformations:

$$[\mathbf{r},1]\ \mathbf{T}_{\text{Model}} = [\mathbf{r}_{\text{world}},1]$$

$$[\mathbf{N},0]\ (\mathbf{T}^{-1}_{\text{Model}})^{\text{T}} = [\mathbf{N}_{\text{world}},d]$$

Camera transformation:

$$[\mathbf{r}_{\text{world}},1]\ \mathbf{T}_{\text{View}} = [\mathbf{r}_{\text{camera}},\ 1]$$

Perspective transformation:

$$[\mathbf{r}_{\text{camera}},1]\ \mathbf{T}_{\text{Persp}} = [\mathbf{r}_{\text{screen}}h,\ h]$$

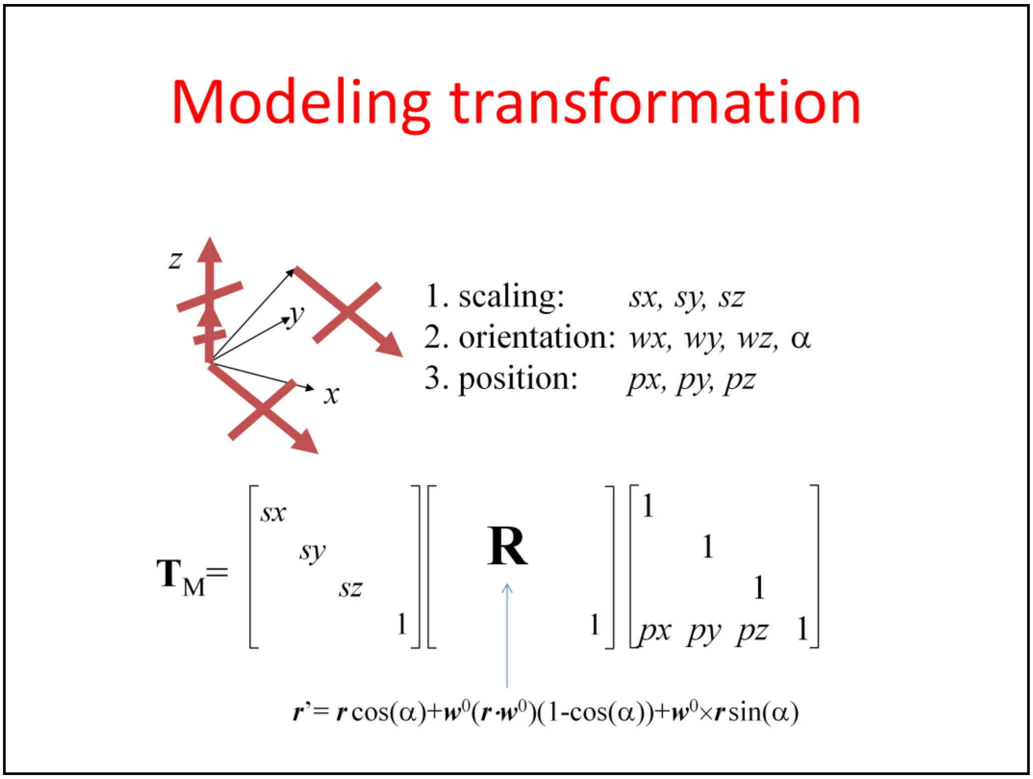
MVP transformation: $\mathbf{T}_{\text{Model}}\mathbf{T}_{\text{View}}\mathbf{T}_{\text{Persp}} = \mathbf{T}_{\text{MVP}}$

Using the modeling transformation, the object is mapped to world coordinates. Recall that the transformation of the shading normals requires the application of the inverse-transpose of the 4x4 matrix, or if the normal is a part of a column vector, we should multiply it with the inverse.

From world coordinates, we go to camera space, where the camera is at the origin and looks at the –z direction. The transformation between world and camera coordinates is a translation and a rotation.

After camera transformation, the next step is perspective transformation, which distorts the objects in a way that the original perspective projection will be equivalent to the parallel projection of the distorted objects. This is a non-affine transformation, so it will not preserve the value of the fourth homogeneous coordinates (which has been 1 so far).

The three transformation matrices (model, camera, perspective) can be concatenated, so a single composite transformation matrix takes us from the reference state directly to normalized screen space.



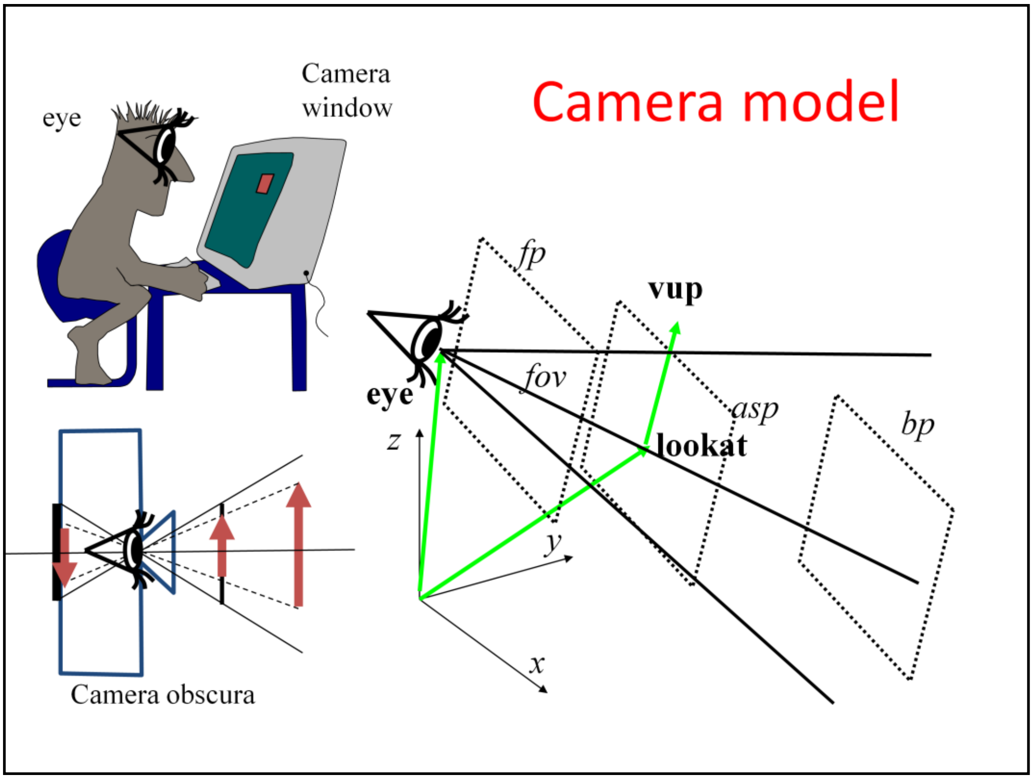
Modeling transformation

1. scaling: *sx, sy, sz*
2. orientation: *wx, wy, wz*, α
3. position: *px, py, pz*

$$\mathbf{T}_{\mathrm{M}}=\begin{bmatrix} sx & & & \\ & sy & & \\ & & sz & \\ & & & 1 \end{bmatrix}\begin{bmatrix} & & & \\ & \mathbf{R} & & \\ & & & \\ & & & 1 \end{bmatrix}\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ px & py & pz & 1 \end{bmatrix}$$

$$\boldsymbol{r}' = \boldsymbol{r}\cos(\alpha)+\boldsymbol{w}^0(\boldsymbol{r}\cdot\boldsymbol{w}^0)(1-\cos(\alpha))+\boldsymbol{w}^0\times\boldsymbol{r}\sin(\alpha)$$

Modeling transformation sets up the object in the virtual world. This means scaling to set its size, then rotation to set its orientation, and finally translation to place it at its position. All three transformations are affine and can be given as a homogeneous transformation matrix. Concatenating the matrices, we obtain a single modeling transformation matrix, which maps the object from its reference state to its actual state.



Camera model

The definition of the camera transformation depends on the parameters of the camera. In computer graphics the camera is the eye position that represents the user in the virtual world and a window rectangle that represents the screen.

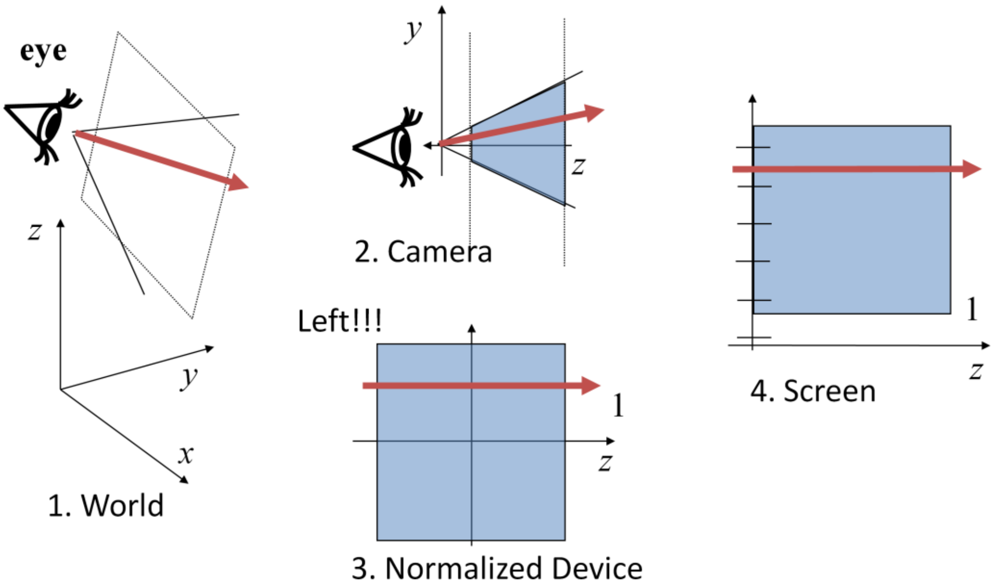
It is often more intuitive to think of a virtual camera as being similar to a real camera. A real camera has a focus point or pin-hole, where the lens is, and a planar film were the image is created in a bottom-up position. In fact, this model is equivalent to the model of the user’s eye and the screen, just the user’s eye should be imagined in the lens position and the film mirrored onto the lens as the screen.

So in both cases, we need to define a virtual eye position and a rectangle in 3D. The **eye** position is a vector in world coordinates. The position of the window is defined by the location of its center, which is called **lookat** point or view reference point. We assume that the **main viewing direction** that is between the eye and the lookat positions is perpendicular to the window. To find the vertical direction of the window, a **view up (vup)** vector needs to be specified. If it is not exactly perpendicular to the viewing direction, then only its perpendicular component is used.

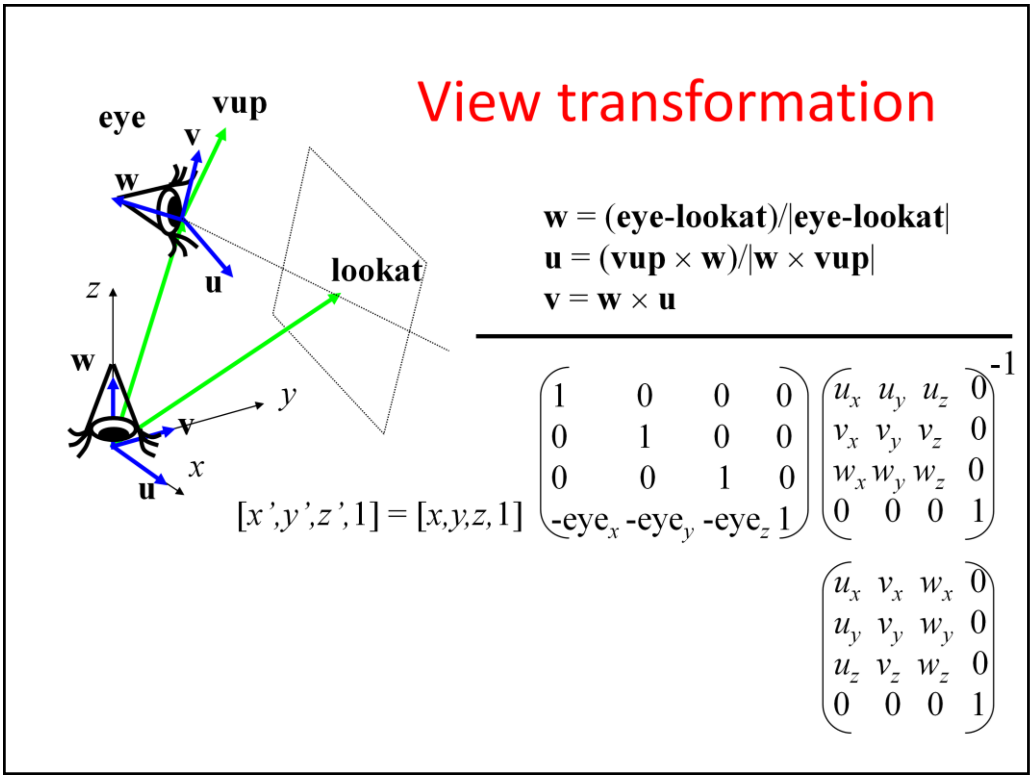
The vectors defined so far specify the window plane and orientation, but not the size of the rectangle. To set the vertical size, the **field of view angle (fov)** is given. For the horizontal size, the **aspect** ratio of the vertical and horizontal window edge sizes should be specified.

Objects being very close to the eye are not visible and leads to numerical inaccuracy. So we also set up a front clipping plane that is parallel to the window and ignore everything that is behind this plane. Similarly, objects that are very far are not visible and may lead to numerical representation problems. So we also introduce a back clipping plane to limit the space for the camera.

From World to Screen



Our goal is to transform the scene from world coordinates to screen coordinates, where visibility determination and projection are trivial. This transformation is built as a sequence of elementary transformations because of pedagogical reasons, but we shall execute all transformations at once, as a single matrix-vector multiplication.



View transformation

w = (**eye**-**lookat**)/|**eye**-**lookat**|

u = (**vup** × **w**)/|**w** × **vup**|

v = **w** × **u**

$$[x',y',z',1] = [x,y,z,1] \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\mathrm{eye}_x & -\mathrm{eye}_y & -\mathrm{eye}_z & 1 \end{pmatrix} \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1}$$

$$\begin{pmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

First we apply a transformation for the scene, including objects and the camera, that moves the camera to the origin and rotates it to make the main viewing be axis –z and the camera's vertical direction be axis y. To find such transformation, we assign an orthonormal basis to the camera so that its first basis vector, u, is the camera's horizontal direction, the second, v, is the vertical direction, and the third, w, is the opposite of the main viewing direction (we reverse the main viewing direction to maintain the right handedness of the system).

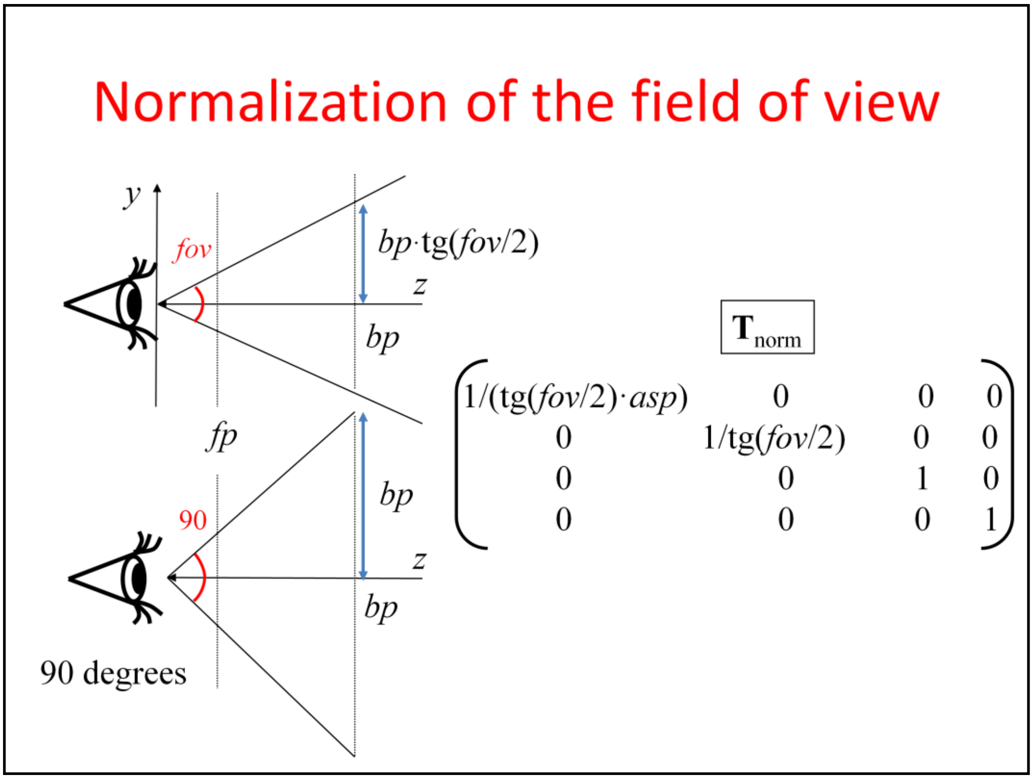
Vector w can be obtained from the main viewing direction by a simple normalization. The application of normalization to get v from vup is also tempting, but simple normalization would not guarantee that basis vector v is orthogonal to basis vector w. So instead of directly computing v from vup, first we obtain u as a vector that is orthogonal to both w and vup. Then, v is computed indirectly through w and u to make it orthogonal to both of them.

The transformation we are looking for is a translation then a rotation. The translation moves the eye position to the origin. The translation has a simple homogeneous linear transformation matrix. Having applied this translation, the orientation should be changed to align vector w with axis z, vector v with axis y, and vector u with axis x. Although this transformation is non-trivial, its inverse that aligns axis x with u, axis y with v, and axis z with w is straightforward.

Its basic idea is that the rows of an affine transformation (fourth column is $[0,0,0,1]^T$) are the images of the three basis vectors and the origin respectively.

So, the transformation of x,y,z axes to u,v,w is the matrix that contains u,v,w as the row vectors of the 3x3 minor matrix of the 4x4 transformation matrix.

As we need the inverse transformation, this matrix needs to be inverted. Such matrices – called orthonormal matrices – are easy to invert, since their transpose is their inverse.

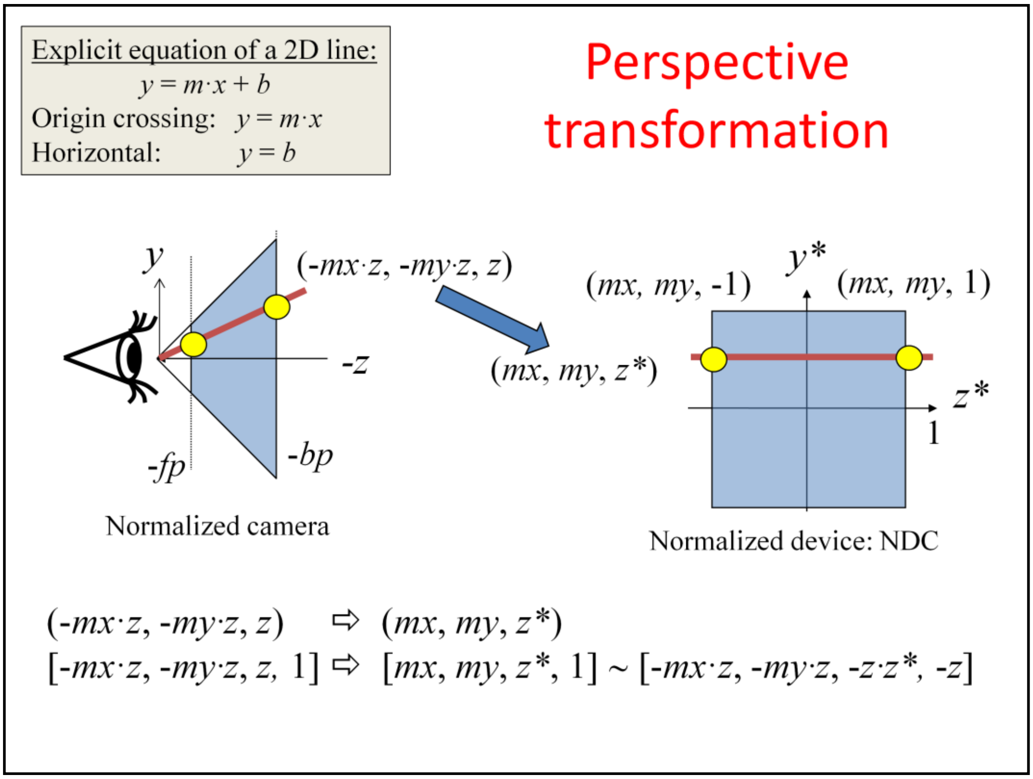


Normalization of the field of view

$$\boxed{\mathbf{T}_{\text{norm}}}$$

$$\begin{pmatrix} 1/(\text{tg}(\mathit{fov}/2)\cdot \mathit{asp}) & 0 & 0 & 0 \\ 0 & 1/\text{tg}(\mathit{fov}/2) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In camera space, the camera is in the origin and the main viewing direction is axis –z. The normalization step distorts the space to make the viewing angle be equal to 90 degrees. This is a scaling along axes y and x. Considering scaling along axis y, before the transformation the top of the viewing pyramid has y coordinate *bp*·tg(*fov*/2), and we expect it to be bp. So, y coordinates must be divided by tg(*fov*/2). Similarly, x coordinates must be divided by tg(*fov*/2)·*asp*.



Perspective transformation

Explicit equation of a 2D line:

$y = m \cdot x + b$

Origin crossing: $y = m \cdot x$

Horizontal: $y = b$

$(-mx \cdot z, -my \cdot z, z) \Rightarrow (mx, my, z^*)$

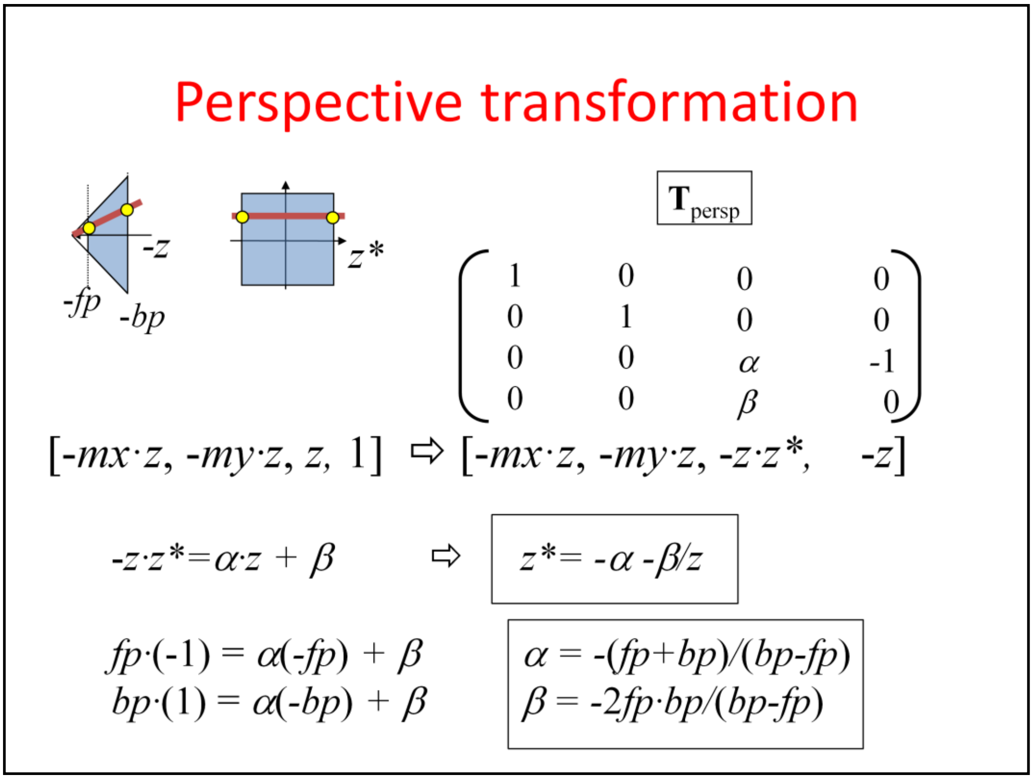
$[-mx \cdot z, -my \cdot z, z, 1] \Rightarrow [mx, my, z^*, 1] \sim [-mx \cdot z, -my \cdot z, -z \cdot z^*, -z]$

Perspective transformation makes the rays meeting in the origin be parallel with axis z, i.e. meeting in infinity (an ideal point at the "end" of axis z).

Additionally, we expect the viewing frustum to be mapped to an axis aligned cube of corner points (-1,-1,-1) and (1,1,1). There are infinitely many solutions for this problem. However, only that solution is acceptable for us which maps lines to lines (and consequently triangles to triangles) since when objects are transformed, we wish to execute the matrix vector multiplication only for the vertices of the triangle mesh and not for every single point (there are infinite of them). Homogeneous linear transformations are known to map lines to lines, so if we can find a homogeneous linear transformation that does the job, we are done. To find the transformation matrix, we consider how a ray should be mapped. A ray can be defined by a line of explicit equation $x=-mx \cdot z$, $y=-my \cdot z$ where coordinate z is a free parameter (mx and my are the slopes of the line). In normalized camera space the slopes are between -1 and 1. We expect this line to be parallel with axis z* after the transformation (z* is the transformed z to resolve ambiguity), so its x and y coordinates should be independent of z.

As x, y must also be in [-1,1], the transformed line is $x^* = mx$, $y^* = my$, and z* is a free parameter.

The mapping from $(x,y,z) = (-mx \cdot z, -my \cdot z, z)$ to (x*,y*,z*)=(mx, my, z^*) cannot be linear in Cartesian coordinates, but is linear (we hope) in homogeneous coordinates. So, we are looking for a linear mapping from $[x,y,z,1] =[-mx \cdot z, -my \cdot z, z, 1]$ to $[x^*,y^*,z^*,1]=[mx, my, z^*, 1]$. To make it simpler, we can exploit the homogeneous property, i.e. the represented point remains the same if all coordinates are multiplied by a non-zero scalar. Let this scalar be -z. So our goal is to find a linear mapping from $[x,y,z,1] =[-mx \cdot z, -my \cdot z, z, 1]$ to to $[x^*,y^*,z^*,1] \sim [-mx \cdot z, -my \cdot z, -z \cdot z^*, -z]$.



Perspective transformation

$$\boxed{\mathbf{T}_{\text{persp}}}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & -1 \\ 0 & 0 & \beta & 0 \end{pmatrix}$$

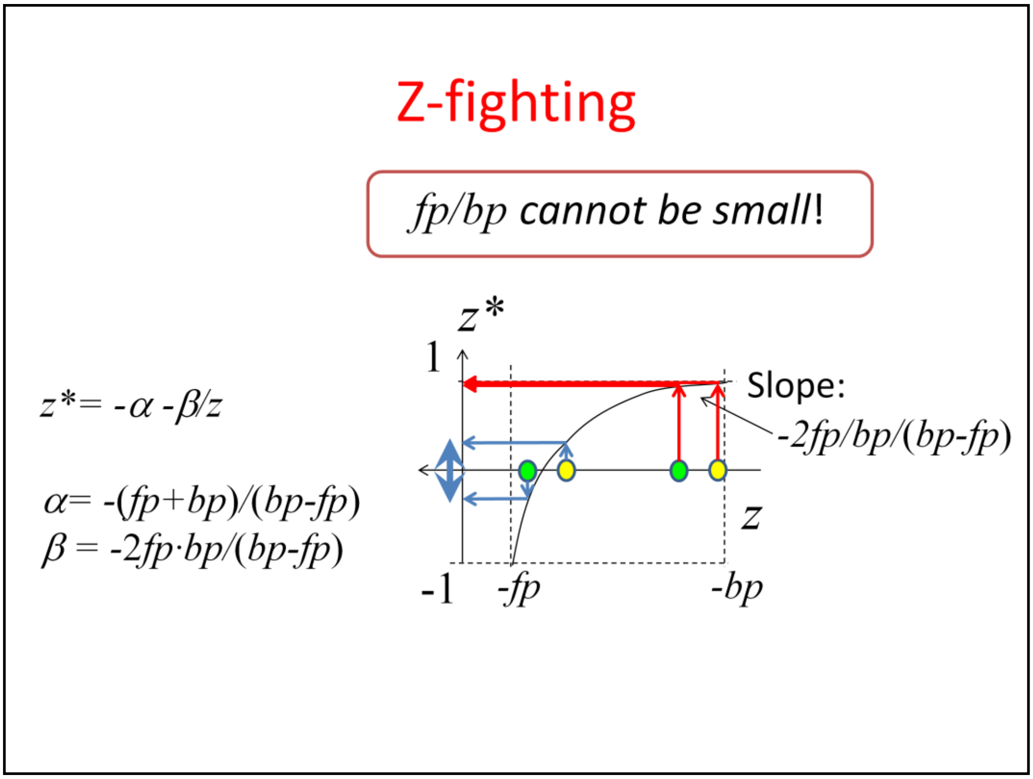
$$[-mx\cdot z,\ -my\cdot z,\ z,\ 1] \Rightarrow [-mx\cdot z,\ -my\cdot z,\ -z\cdot z^*,\quad -z]$$

$$-z\cdot z^* = \alpha\cdot z + \beta \quad \Rightarrow \quad \boxed{z^* = -\alpha - \beta/z}$$

$$fp\cdot(-1) = \alpha(-fp) + \beta$$
$$bp\cdot(1) = \alpha(-bp) + \beta$$

$$\boxed{\begin{array}{l} \alpha = -(fp+bp)/(bp-fp) \\ \beta = -2fp\cdot bp/(bp-fp) \end{array}}$$

A homogeneous linear transformation is a 4x4 matrix, i.e. sixteen scalars, which need to be found. The requirement is that for arbitrary *mx, my, z*, when multiplying with [-*mx·z*, -*my·z*, *z,* 1], the result must be [-*mx·z*, -*my·z*, -*z·z**, -*z*]. The first two elements are kept for arbitrary *mx, my, z*, which is possible if the first two colums of the matrix are [1,0,0,0] and [0,1,0,0]. As mx and my do not affect the third and the fourth elements in the result, the corresponding matrix element must be zero. The fourth element of the result is –z, so the fourth column is [0,0,-1, 0]. We are left with only two unknown parameters alpha and beta. They can be found by considering the requirements that the entry point that is the intersection of the ray and the front clipping plane is mapped to z*=-1 and the exit point to z*=1.



Z-fighting

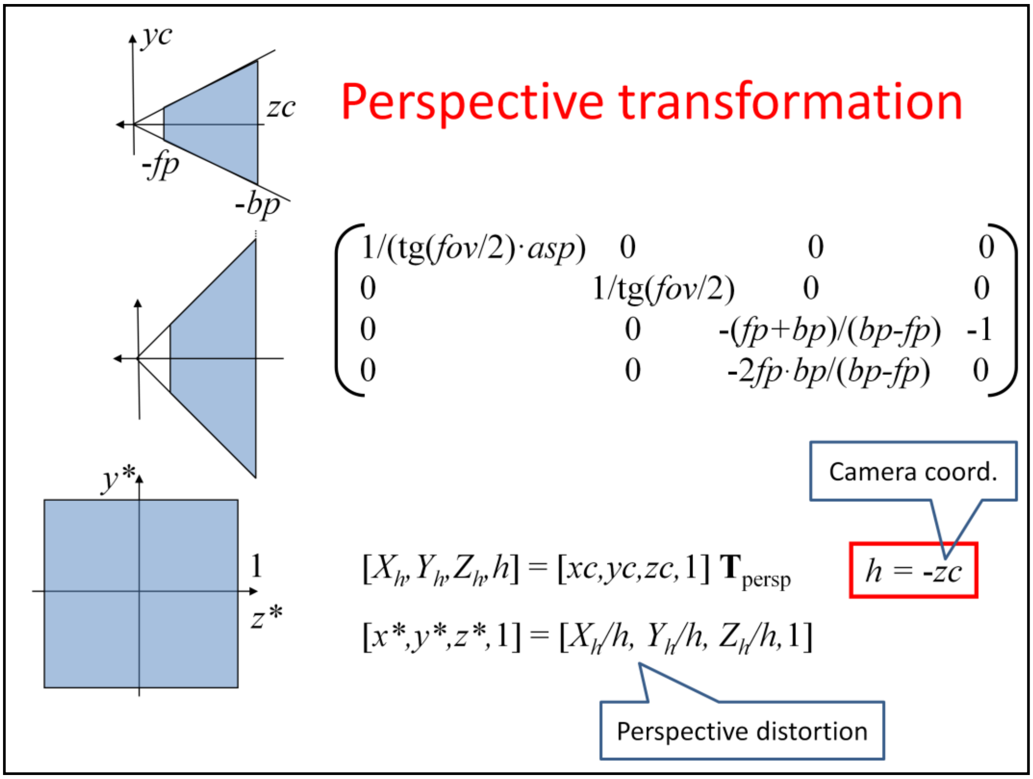
fp/bp cannot be small!

$$z^* = -\alpha - \beta/z$$

$$\alpha = -(fp+bp)/(bp-fp)$$
$$\beta = -2fp\cdot bp/(bp-fp)$$

Note that the expression of z* as a function of z is not a linear (which we knew from the very beginning), but a reciprocal function. Recall that this 1/x function changes quickly where x is small but will be close to constant where x is large. Value z* is used to determine visibility. To determine visibility robustly, the difference of z* for two points must be large enough (otherwise comparison fails due to numerical inaccuracies and number representation limitations). This is not the case if β is small and z is large, e.g. when z≈-bp approximately, i.e. when $\beta/z \approx 2fp/(bp-fp)$. If *bp* is much greater than *fp*, then $\beta/z \approx 2fp/(bp-fp) \approx 2fp/bp$ is very small, prohibiting to robustly distinguish two occluding surfaces.

Never specify *fp* and *bp* such that *fp/bp* is too small (e.g. less than 0.01). If you do, occluded surfaces will randomly show up because of numerical inaccuracy. This phenomenon is called **z-fighting**.



Perspective transformation

$$\begin{pmatrix} 1/(\mathrm{tg}(fov/2)\cdot asp) & 0 & 0 & 0 \\ 0 & 1/\mathrm{tg}(fov/2) & 0 & 0 \\ 0 & 0 & -(fp+bp)/(bp-fp) & -1 \\ 0 & 0 & -2fp\cdot bp/(bp-fp) & 0 \end{pmatrix}$$

Camera coord.

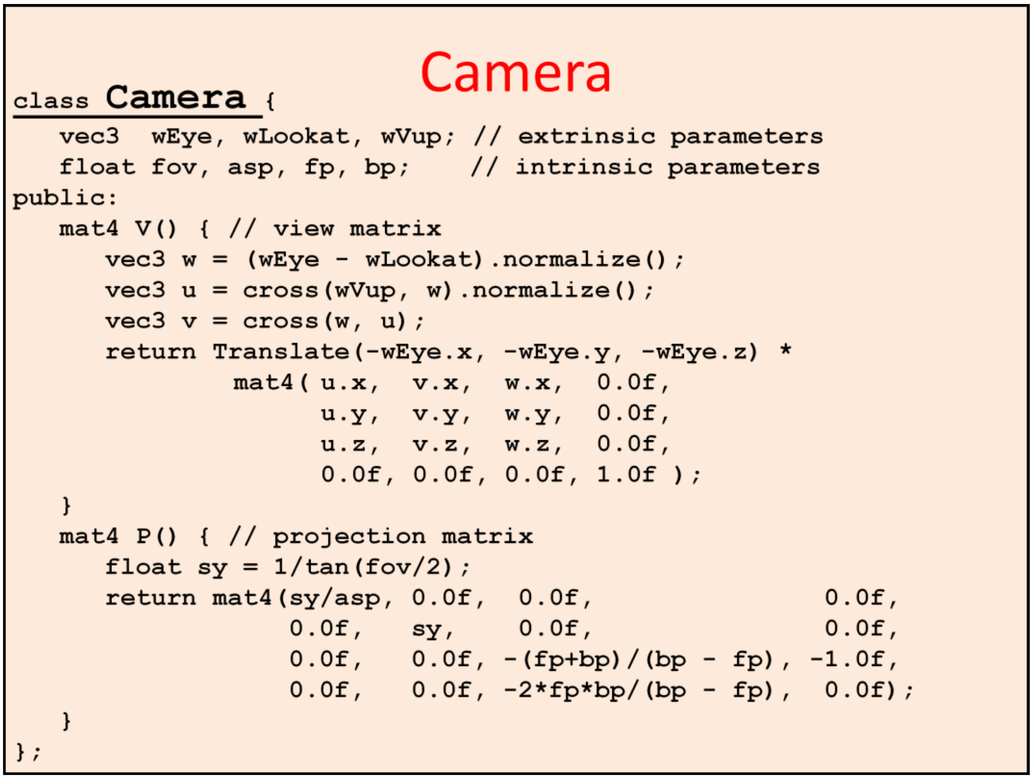
$$[X_h, Y_h, Z_h, h] = [xc, yc, zc, 1]\,\mathbf{T}_{\mathrm{persp}} \qquad \boxed{h = -zc}$$

$$[x^*, y^*, z^*, 1] = [X_h/h,\ Y_h/h,\ Z_h/h, 1]$$

Perspective distortion

Normalization and perspective transformation are usually combined and the composed transformation is set directly.

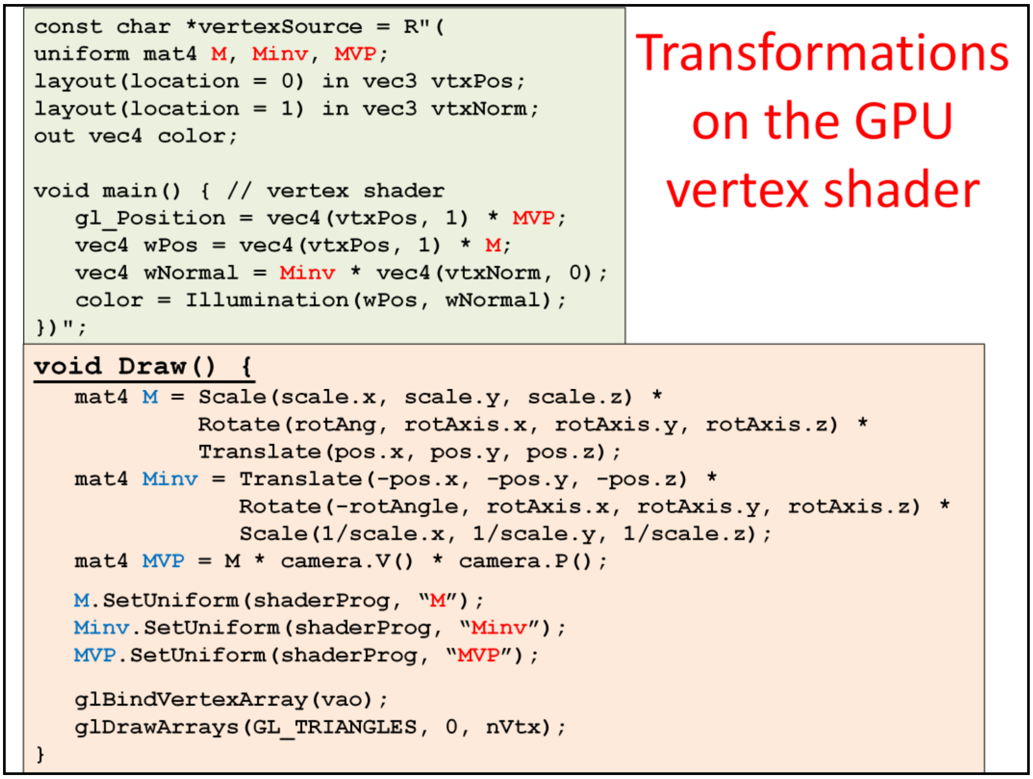
It is worth noting that this transformation sets the fourth homogeneous coordinate to the camera coordinate depth value. It is also notable that this transformation maps the eye ([0,0,0,1] in homogeneous coordinates) to the ideal point of axis z, i.e. to [0,0, -2*fp* ·*bp*/(*bp*-*fp*), 0] ~[0,0, 1, 0].



Camera

```
class Camera {
   vec3  wEye, wLookat, wVup; // extrinsic parameters
   float fov, asp, fp, bp;    // intrinsic parameters
public:
   mat4 V() { // view matrix
      vec3 w = (wEye - wLookat).normalize();
      vec3 u = cross(wVup, w).normalize();
      vec3 v = cross(w, u);
      return Translate(-wEye.x, -wEye.y, -wEye.z) *
              mat4( u.x,  v.x,  w.x,  0.0f,
                    u.y,  v.y,  w.y,  0.0f,
                    u.z,  v.z,  w.z,  0.0f,
                    0.0f, 0.0f, 0.0f, 1.0f );
   }
   mat4 P() { // projection matrix
      float sy = 1/tan(fov/2);
      return mat4(sy/asp, 0.0f,  0.0f,               0.0f,
                  0.0f,   sy,    0.0f,               0.0f,
                  0.0f,   0.0f, -(fp+bp)/(bp - fp), -1.0f,
                  0.0f,   0.0f, -2*fp*bp/(bp - fp),  0.0f);
   }
};
```

Our 3D Camera class stores parameters need to define a camera, and implement two transformation functions. Transformation V is the view transformation that takes a point form world space to camera space, and transformation P is the projection or perspective transformation that takes a point from camera space to normalized device space.

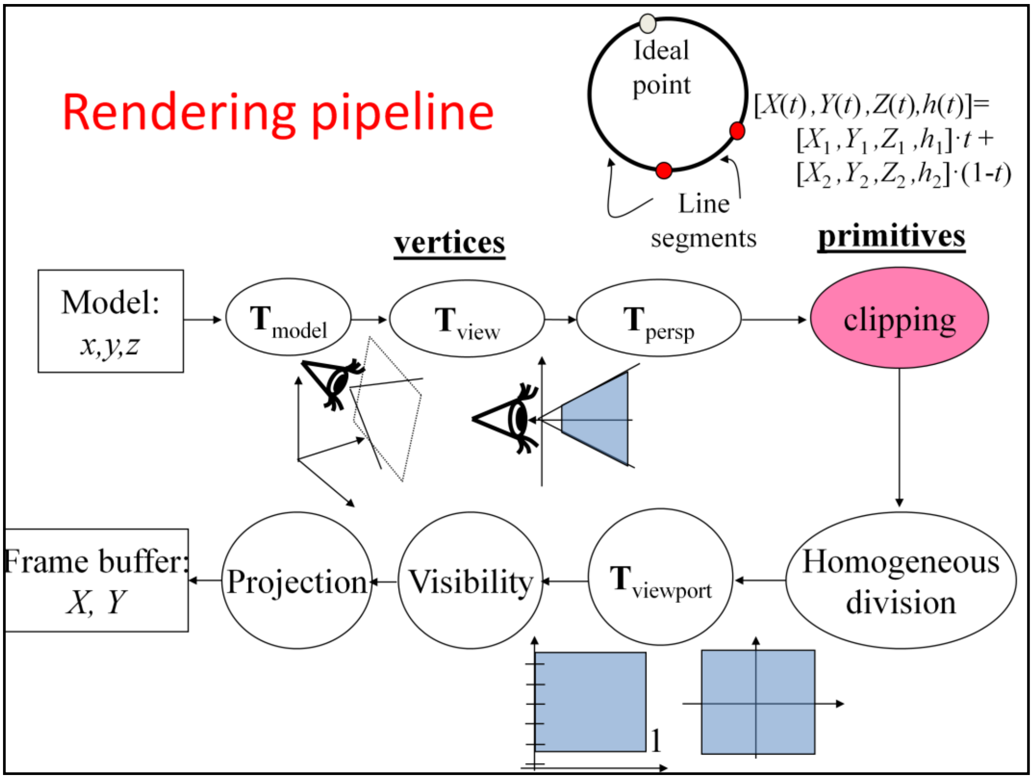


Transformations on the GPU vertex shader

```
const char *vertexSource = R"(
uniform mat4 M, Minv, MVP;
layout(location = 0) in vec3 vtxPos;
layout(location = 1) in vec3 vtxNorm;
out vec4 color;

void main() { // vertex shader
   gl_Position = vec4(vtxPos, 1) * MVP;
   vec4 wPos = vec4(vtxPos, 1) * M;
   vec4 wNormal = Minv * vec4(vtxNorm, 0);
   color = Illumination(wPos, wNormal);
})";
```

```
void Draw() {
   mat4 M = Scale(scale.x, scale.y, scale.z) *
            Rotate(rotAng, rotAxis.x, rotAxis.y, rotAxis.z) *
            Translate(pos.x, pos.y, pos.z);
   mat4 Minv = Translate(-pos.x, -pos.y, -pos.z) *
               Rotate(-rotAngle, rotAxis.x, rotAxis.y, rotAxis.z) *
               Scale(1/scale.x, 1/scale.y, 1/scale.z);
   mat4 MVP = M * camera.V() * camera.P();

   M.SetUniform(shaderProg, “M”);
   Minv.SetUniform(shaderProg, “Minv”);
   MVP.SetUniform(shaderProg, “MVP”);

   glBindVertexArray(vao);
   glDrawArrays(GL_TRIANGLES, 0, nVtx);
}
```

Transformations are set on the CPU side but are applied to points by the GPU. Transformations are uniform variables of the vertex shader, which are set from the CPU program and used by the vertex shader. We always need the MVP, i.e. model-view-projection transformation that is the concatenation of the modeling transformation and the two phases of the camera transformation. If illumination is also computed in world coordinate system, object should also be transformed there. For points, we need the modeling transformation M, for normal vectors its inverse. Note that normal vectors stand on the right side of the matrix, i.e. they form column vectors.



Rendering pipeline

Ideal point

$$[X(t), Y(t), Z(t), h(t)] = [X_1, Y_1, Z_1, h_1] \cdot t + [X_2, Y_2, Z_2, h_2] \cdot (1-t)$$

Line segments

vertices

primitives

Model: x,y,z

$\mathbf{T}_{\text{model}}$

$\mathbf{T}_{\text{view}}$

$\mathbf{T}_{\text{persp}}$

clipping

Homogeneous division

$\mathbf{T}_{\text{viewport}}$

Visibility

Projection

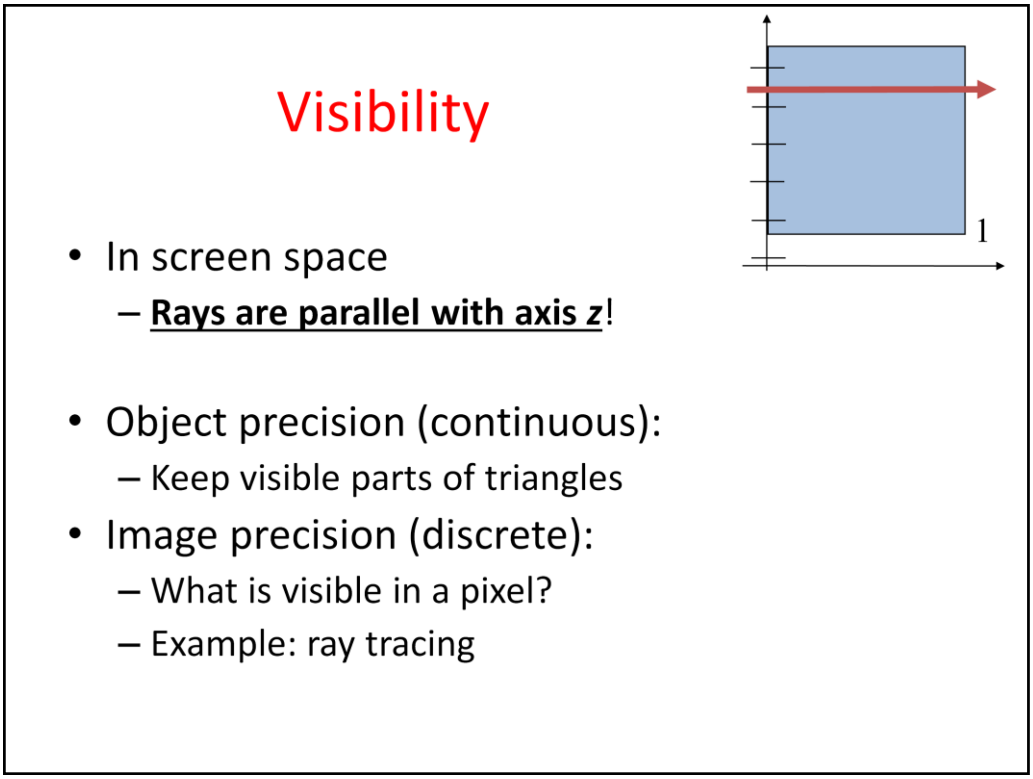
Frame buffer: X, Y

1

Clipping must be done before the homogeneous division, preferably in a coordinate system where it is the simplest. The optimal choice is the normalized device coordinates where the view frustum is a cube. However, clipping must be done before the homogeneous division. This might be surprising but becomes clear if we consider the topology of a projective line and the interpretation of a line segment. A projective line is like a circle, the ideal point attaches the two “endpoints”. As on a circle, two points do not unambiguously identify an arc (there are two possible arcs), on a projective line, two points may define two complementer “line segments” (one of them looks as two half lines). To resolve this ambiguity, we specify endpoints with positive fourth homogeneous coordinates, and define the line segment as a convex combination of two endpoints. If fourth coordinates h are positive for both end points, then the interpolated h cannot be zero, so this line segment does not contain the ideal point (it is a “normal line segment”). Affine transformations usually do not alter the fourth homogeneous coordinate, so “normal line segments” will remain normal.

Perspective transformation may map a line segment to a line segment that contains the ideal point, which is clearly indicated in the different signs of the fourth homogeneous coordinates of the two endpoints. So, if the two h coordinates have the same sign, then the line segment is normal. If the two h coordinates have different sign, then the line segment is wrapped around (it contains the ideal point so we would call it two half lines and not a line segment).

One way to solve this problem is to clip away everything that is behind the front clipping plane. This clipping must be executed before the homogeneous division since during this operation we lose the information regarding the sign of the fourth homogeneous coordinate.



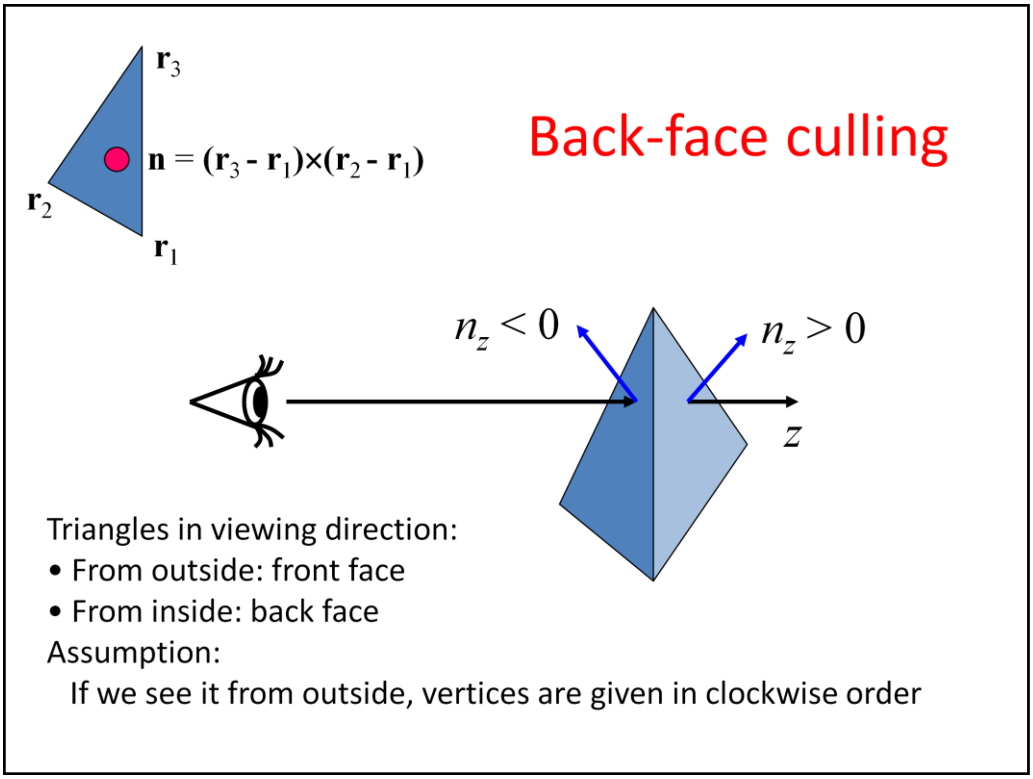
Visibility

- In screen space
  - **Rays are parallel with axis *z***!
- Object precision (continuous):
  - Keep visible parts of triangles
- Image precision (discrete):
  - What is visible in a pixel?
  - Example: ray tracing

We need solid rendering which produces filled polygons and also considers occlusions. Many polygons may be mapped onto the same pixel. We should find that polygon (and its color), which occludes the others, i.e. which are the closest to the eye position. This calculation is called **visibility determination.**

We have to emphasize that visibility is determined in the screen coordinate system where rays are parallel with axis z and the x, y coordinates are the physical pixel coordinates. We made all the complicated looking homogeneous transformations just for this, to calculate visibility and projection in a coordinate system where these operations are simple.

There are many visibility algorithms (from which we shall discuss only 1.5 :-). The literature classifies them as object space (aka object precision or continuous) and screen space (aka image precision or discrete). Object space visibility algorithms find the visible portions of triangles in the form of (possibly clipped and subdivided) triangles, independently of the resolution of the image. Screen space algorithms exploit the fact that the image has finite precision and determine the visibility just at finite number of points, usually through the centers of the pixels. Recall that ray tracing belongs to the category of image precision algorithms.



Back-face culling

$\mathbf{r}_3$
$\mathbf{n} = (\mathbf{r}_3 - \mathbf{r}_1)\times(\mathbf{r}_2 - \mathbf{r}_1)$
$\mathbf{r}_2$
$\mathbf{r}_1$
$n_z < 0$
$n_z > 0$
z

Triangles in viewing direction:

- From outside: front face
- From inside: back face

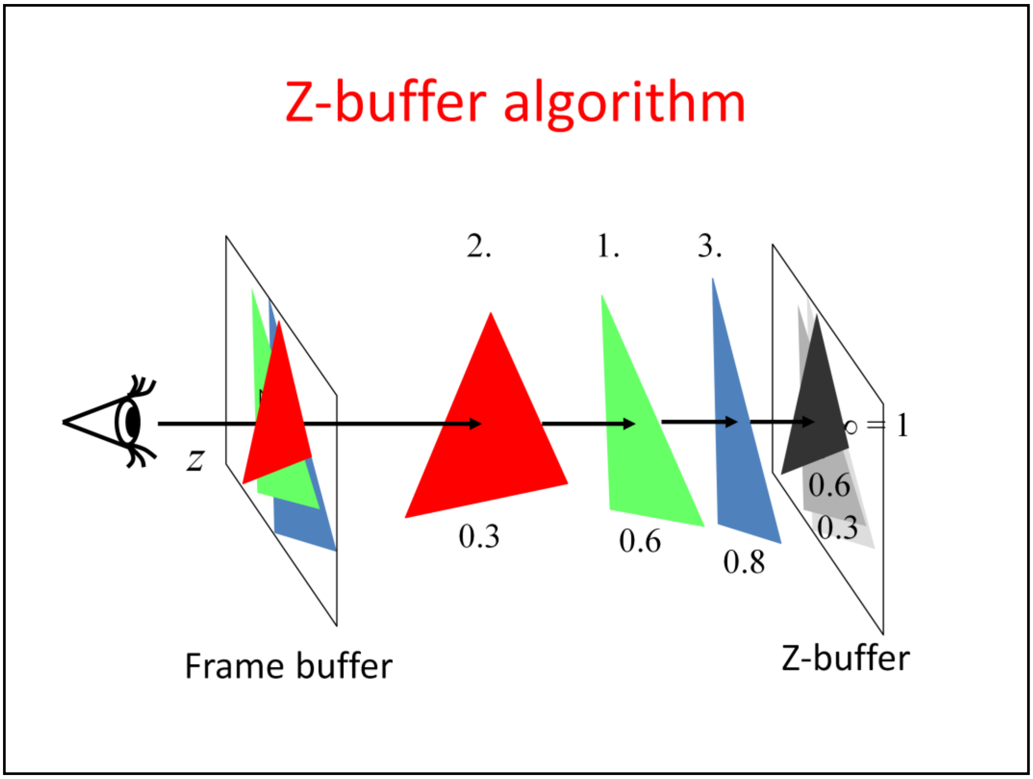
Assumption:

If we see it from outside, vertices are given in clockwise order

First an object space method is presented, which gives only partial solution.

Assume that the triangle mesh is the boundary of a valid 3D object! In this case, we can see only one side or face of each boundary polygon. The potentially visible face is called **front-face**, the other face or side where the polygon is ”glued” to its object is called **back-face**. A back-face is never visible since the object itself always occludes it. So, when a polygon shows its back-face to the camera, we can simply ignore it since we know that there are other polygons of the same object, which will occlude it. To help the determination whether a face is front or back, we use a coding scheme that must be set during the modeling of the geometry (or during tessellation). In theory, triangle or polygon vertices can be specified either in clock-wise or counter-clock-wise order. Let us use this ordering to indicate which face is glued to the object. For example, we can state that vertices must be specified in clock-wise order when the object is seen from outside, i.e. when we see the front-face of the polygon. Computing a geometric normal with $\mathbf{n} = (\mathbf{r}_3 - \mathbf{r}_1)\times(\mathbf{r}_2 - \mathbf{r}_1)$, the clock-wise order means that **n** points towards the viewer. As in screen coordinates the viewing rays are parallel with axis z, it is easy to decide whether a vector points towards the viewer, simply the z coordinate of **n** must be negative.

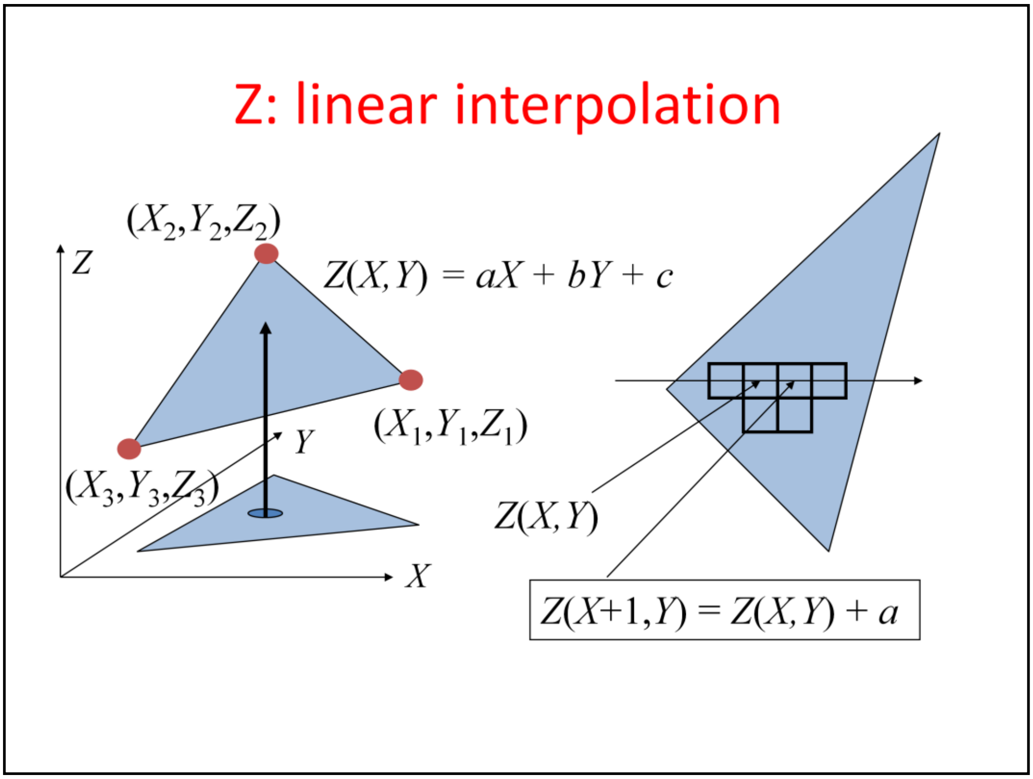
With back-face culling, we can get rid of the surely non-visible polygons, which always helps. However, front-faces may also occlude each other, which should be resolved by another algorithm. Still, back-face culling is worth applying, since it roughly halves the number of the potentially visible polygons.



Z-buffer algorithm

And now let us meet the far most popular visibility algorithm of incremental image synthesis. Ray tracing considers pixels one by one and for a single pixel, it checks each object for intersection while always keeping the first intersection found so far. In incremental rendering, we would like to exploit the similarities of different points on an object (on a triangle), so instead of processing every pixel independently, we take an object centric approach. Let us swap the order of the for loops of ray tracing and take triangles one by one and for a single triangle, find visibility rays intersecting it, always keeping the first intersection in every pixel. As all pixels are processed simultaneously, instead of a single minimum ray parameter, we need to maintain as many ray parameters as pixels the viewport has. In screen coordinates, the ray parameter, i.e. the distance of the intersection, is represented by the z coordinate of the intersection point. Therefore, the array storing the ray parameters of the intersections is called the z-buffer or depth buffer.

The z-buffer algorithm initializes the depth buffer to the largest possible value, since we are looking for a minimum. In screen coordinates, the maximum of z coordinates is 1 (the back clipping plane is here). The algorithm takes triangles one-by-one in an arbitrary order. A given triangle is projected onto the screen and rasterized, i.e. we visit those pixels that are inside the triangle's projection (this step is equivalent to identifying those rays which intersect the given object). At a particular pixel, the z coordinate of the point of the triangle visible in this pixel is computed (this is the same as ray triangle intersection), and the triangle's z coordinate is compared to the z value stored in the z buffer at this pixel. If the triangle's z coordinate is smaller, then the triangle point is closer to the front clipping plane at this pixel than any of the triangles processed so far, so the triangle's color is written into the frame buffer and its z coordinate to the depth buffer. Thus, the frame buffer and the depth buffer always store the color and depth of that triangles which are visible from the triangles processed so far.



Z: linear interpolation

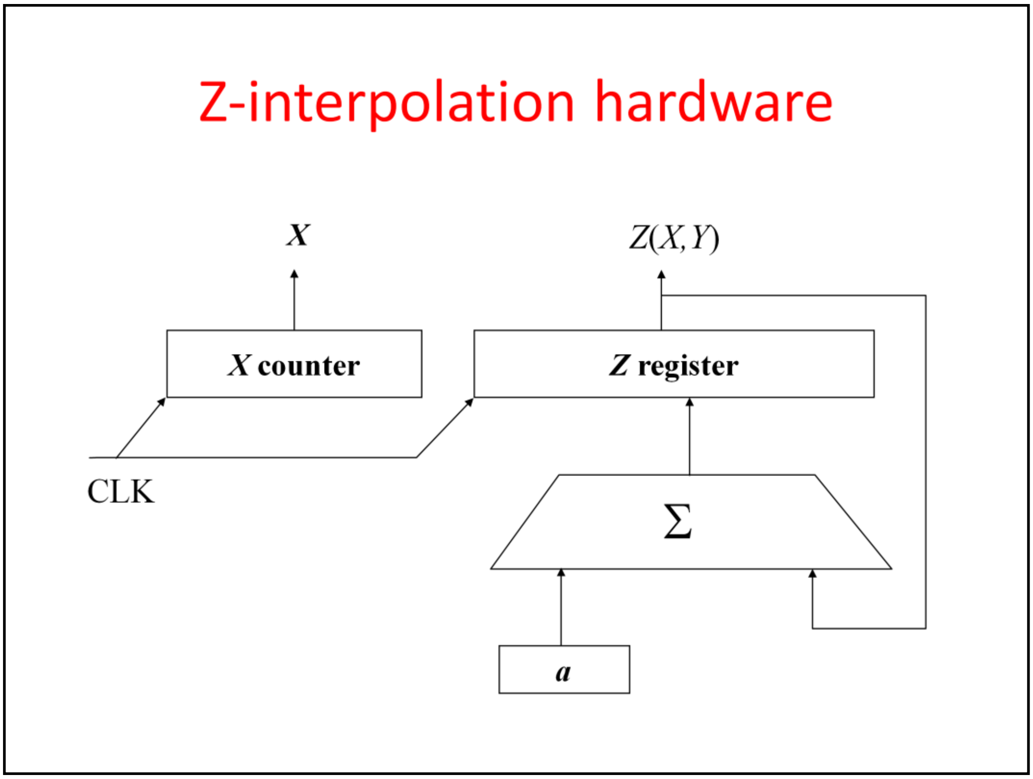
$$Z(X,Y) = aX + bY + c$$

$$\boxed{Z(X+1,Y) = Z(X,Y) + a}$$

Projected triangles are rasterized by visiting scan lines (i.e. rows of pixels) and determining the intersection of this horizontal line and the edges of the triangle. Pixels of X coordinates that are in between the X coordinates of the two edge intersections are those where the triangle is potentially visible.

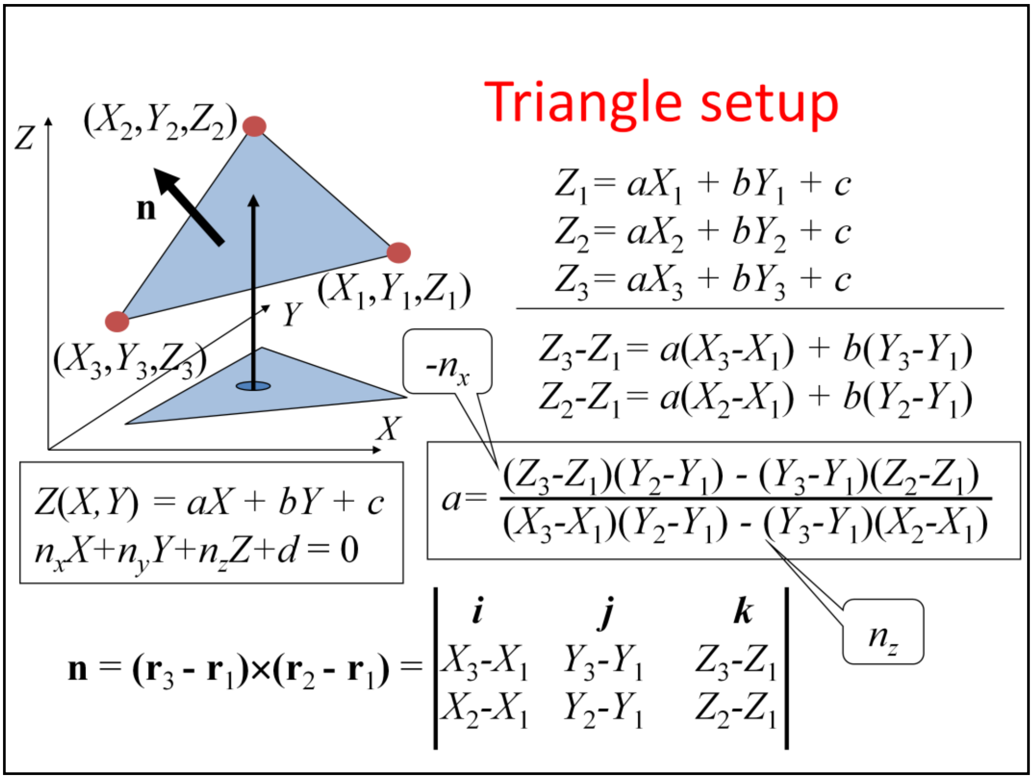
Pixels in this row are visited from left to right, and the z coordinate of the triangle point whose projection is this pixel center is calculated. This z coordinate is compared to the value of the depth buffer.

The first question is how the Z coordinates of the triangle points are computed for every pixel of coordinates X and Y. The triangle is on a plane so X, Y, Z satisfy the plane equation, i.e. a linear equation, so Z will also be a linear function of pixel coordinates X and Y. The evaluation of this linear function requires two multiplications and two additions, which are too much if we have just a few nanoseconds for this operation. To speed up the process, **the incremental principle** can be exploited. As we visit pixels in the order of coordinate X, when a pixel of coordinates X+1,Y is processed, we have the solution, Z, for the previous pixel. It is easier to compute only the increment than obtaining Z from scratch. Indeed, the difference between Z(X+1,Y) and Z(X,Y) is constant so the new depth can be obtained from the previous one as a single addition with a constant.



Z-interpolation hardware

Such an incremental algorithm is easy to be implemented directly in hardware. A counter increments its value to generate coordinate X for every clock cycle. A register that stores the actual Z coordinate in fixed point, non-integer format (note that increment *a* is usually not an integer). This register is updated with the sum of its previous value and *a* for every clock cycle.



Triangle setup

$$Z_1 = aX_1 + bY_1 + c$$
$$Z_2 = aX_2 + bY_2 + c$$
$$Z_3 = aX_3 + bY_3 + c$$

$$Z_3 - Z_1 = a(X_3 - X_1) + b(Y_3 - Y_1)$$
$$Z_2 - Z_1 = a(X_2 - X_1) + b(Y_2 - Y_1)$$

$$Z(X,Y) = aX + bY + c$$
$$n_x X + n_y Y + n_z Z + d = 0$$

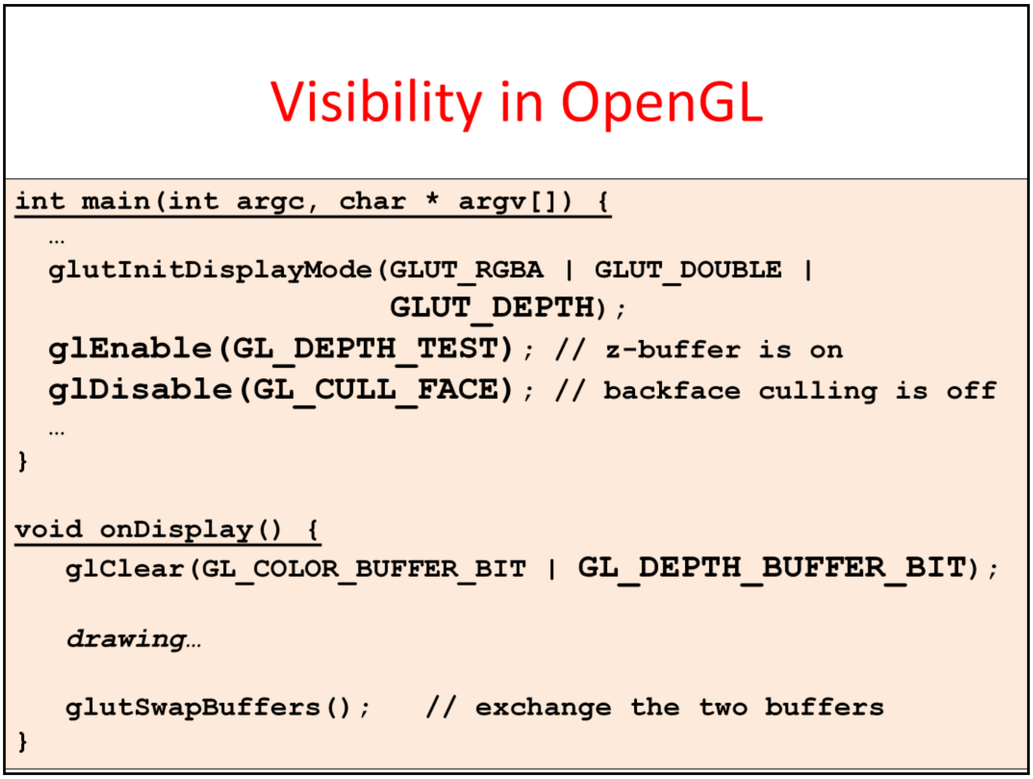
$$a = \frac{(Z_3 - Z_1)(Y_2 - Y_1) - (Y_3 - Y_1)(Z_2 - Z_1)}{(X_3 - X_1)(Y_2 - Y_1) - (Y_3 - Y_1)(X_2 - X_1)}$$

$$\mathbf{n} = (\mathbf{r}_3 - \mathbf{r}_1)\times(\mathbf{r}_2 - \mathbf{r}_1) = \begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ X_3 - X_1 & Y_3 - Y_1 & Z_3 - Z_1 \\ X_2 - X_1 & Y_2 - Y_1 & Z_2 - Z_1 \end{vmatrix}$$

The final problem is how Z increment *a* is calculated. One way of determining it is to satisfy the interpolation constraints at the three vertices.

The other way is based on the recognition that we work with the plane equation where X,Y,Z coordinates are multiplied by the coordinates of the plane’s normal. The normal vector, in turn, can be calculated as a cross product of the edge vectors.

Note that the coordinates of the screen space normal are needed not only here, but also in back-face culling. So, triangle set up is also responsible for the front-face, back-face classification.

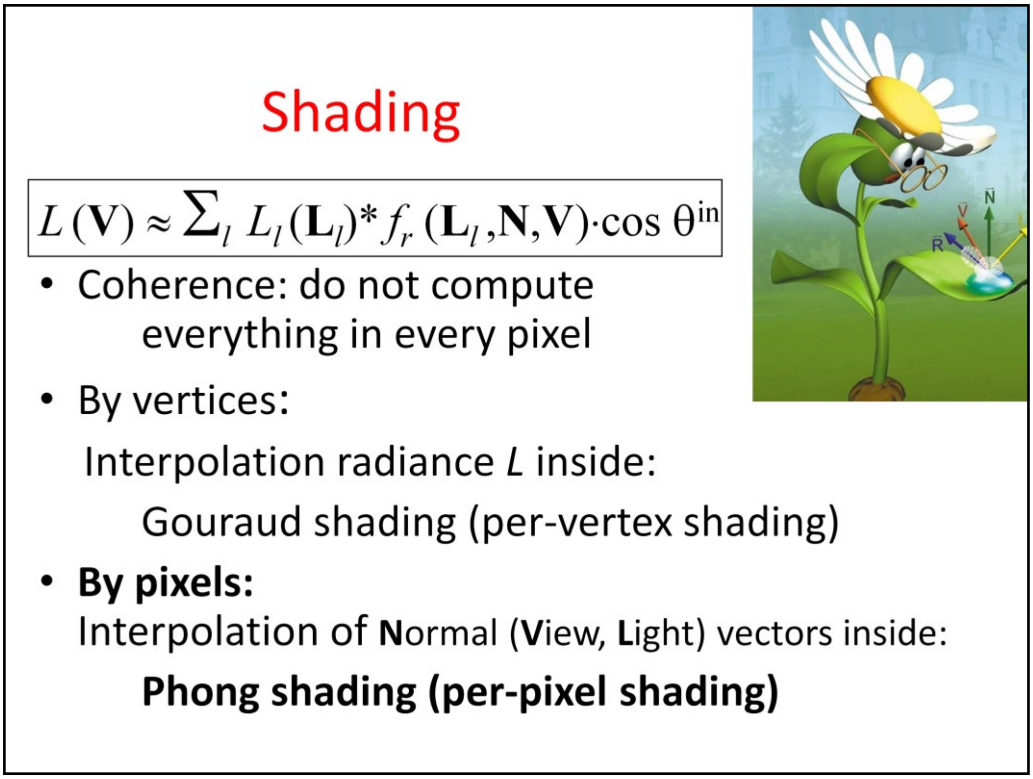


Visibility in OpenGL

```
int main(int argc, char * argv[]) {
  …
  glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE |
                      GLUT_DEPTH);
  glEnable(GL_DEPTH_TEST); // z-buffer is on
  glDisable(GL_CULL_FACE); // backface culling is off
  …
}

void onDisplay() {
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   drawing…

   glutSwapBuffers();   // exchange the two buffers
}
```

Back face culling and the z-buffer algorithm are executed by the GPU in its fixed function pipeline. However, we can enable or disable their operation. If we want to use z-buffering, in glutInitDisplayMode, we should specify that a pixel should have depth channel storing the depth values. Then, z-buffering can be enabled with **glEnable(GL_DEPTH_TEST)** and disabled with glDisable(GL_DEPTH_TEST). Back face culling can also be enabled or disabled by switching GL_CULLING FACE.

If we use depth buffering, the initialization of the z-buffer should be done at the beginning of the drawing cycle by telling glClear that not only the frame buffer, but also the depth buffer should be initialized.



Shading

$$\boxed{L(\mathbf{V}) \approx \sum_l L_l(\mathbf{L}_l)* f_r(\mathbf{L}_l,\mathbf{N},\mathbf{V})\cdot\cos\theta^{\mathrm{in}}}$$

- Coherence: do not compute everything in every pixel
- By vertices:

  Interpolation radiance *L* inside:

  Gouraud shading (per-vertex shading)
- **By pixels:**

  Interpolation of **N**ormal (**V**iew, **L**ight) vectors inside:

  Phong shading (per-pixel shading)

$\vec{V}$
$\vec{N}$
$\vec{R}$

When a triangle point turns out to be visible (at least among the triangles processed so far), its color should be computed and written into the frame buffer.

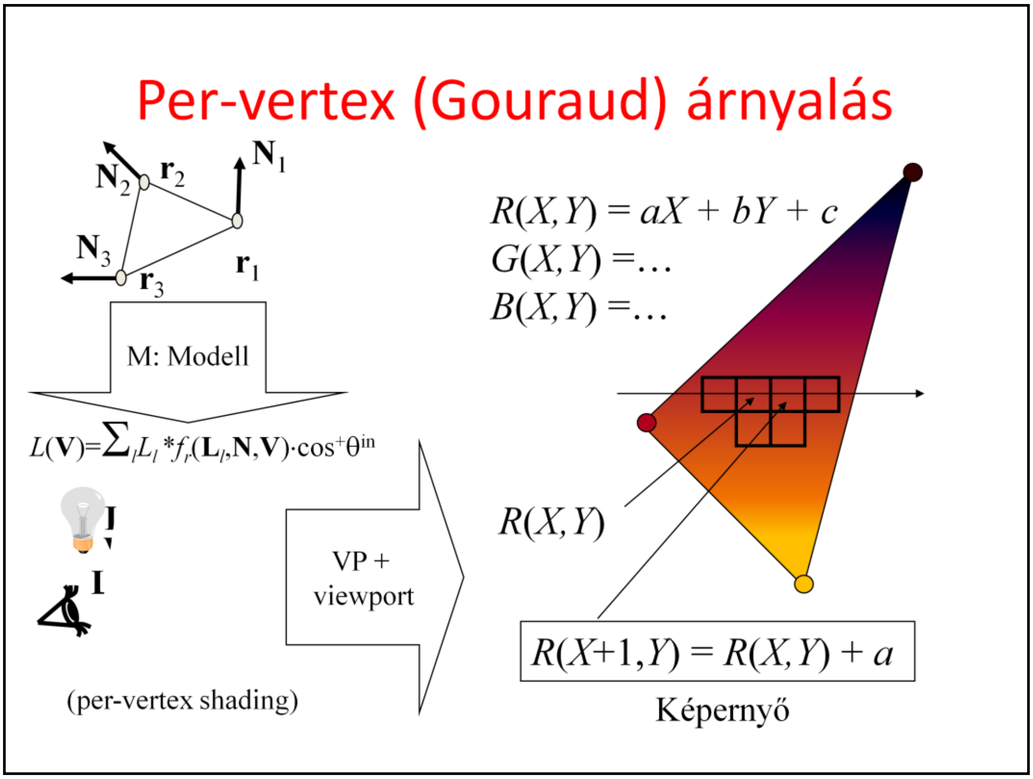
According to optics, the color is the radiance calculated in the direction of the eye, which is the sum of the contributions of abstract light sources in the local illumination model. For a single light source, the radiance is the light source intensity times the BRDF times the geometry factor.

In incremental rendering, we try to reuse calculation done in other pixels, so the evaluation of this formula for different pixels shares computations.

For example, we can evaluate this illumination formula only for the three vertices and apply linear interpolation in between the vertices (Gouraud shading or per-vertex shading).

Or, while the illumination formula is evaluated at every pixel, the vectors needed for the calculation are interpolated from the vectors available at the vertices (Phong shading or per-pixel shading).

Per-vertex (Gouraud) árnyalás



$$L(\mathbf{V})=\sum_l L_l * f_r(\mathbf{L}_l,\mathbf{N},\mathbf{V})\cdot\cos^+\theta^{in}$$

(per-vertex shading)

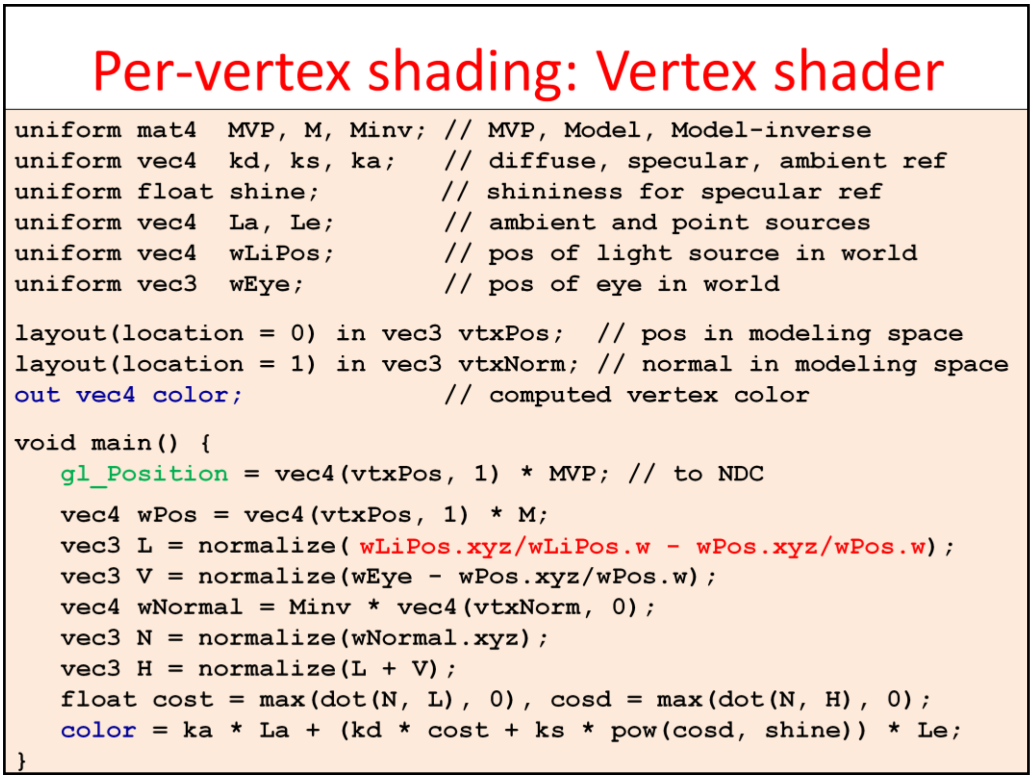
$$R(X,Y) = aX + bY + c$$
$$G(X,Y) = \ldots$$
$$B(X,Y) = \ldots$$

$$\boxed{R(X+1,Y) = R(X,Y) + a}$$

The radiance can be computed in world coordinates since here we have everything together needed by the illumination calculation, including the illuminated objects, light sources, and the camera. Alternatively, radiance can also be computed in camera space since the illumination formula depends on angles (e.g. between the illumination direction and the surface normal) and distances (in case of point light sources), and the transformation between world space and camera space is angle and distance preserving (congruence), so we obtain the same results in the two coordinate systems. However, radiance computation must not be postponed to later stages of the pipeline (e.g. normalized camera space, normalized device space, or screen space) since the mapping to these coordinate systems may modify angles.

After tessellation, the object is a set of triangles with vertices and shading normals. These are transformed to world space (or to camera space) by multiplying the vertices by the modeling transform (or by the modeling and camera transforms) and the normals by the inverse-transport of this matrix.

In this space, the view and light directions are computed at the triangle vertices. If per-vertex shading is applied, these vectors are immediately inserted into the illumination formula obtaining the reflected radiance at the vertices. The triangle vertices with their computed colors are mapped to screen space, where rasterization takes place. The color of the internal pixels is computed by linear interpolation of the r,g,b values of the colors at the vertices. This is very similar to the interpolation of depth values.



Per-vertex shading: Vertex shader

```
uniform mat4  MVP, M, Minv; // MVP, Model, Model-inverse
uniform vec4  kd, ks, ka;   // diffuse, specular, ambient ref
uniform float shine;        // shininess for specular ref
uniform vec4  La, Le;       // ambient and point sources
uniform vec4  wLiPos;       // pos of light source in world
uniform vec3  wEye;         // pos of eye in world

layout(location = 0) in vec3 vtxPos;  // pos in modeling space
layout(location = 1) in vec3 vtxNorm; // normal in modeling space
out vec4 color;             // computed vertex color

void main() {
   gl_Position = vec4(vtxPos, 1) * MVP; // to NDC

   vec4 wPos = vec4(vtxPos, 1) * M;
   vec3 L = normalize( wLiPos.xyz/wLiPos.w - wPos.xyz/wPos.w);
   vec3 V = normalize(wEye - wPos.xyz/wPos.w);
   vec4 wNormal = Minv * vec4(vtxNorm, 0);
   vec3 N = normalize(wNormal.xyz);
   vec3 H = normalize(L + V);
   float cost = max(dot(N, L), 0), cosd = max(dot(N, H), 0);
   color = ka * La + (kd * cost + ks * pow(cosd, shine)) * Le;
}
```

Let us see the vertex shader of the per-vertex shading approach. As its name suggests, illumination computation takes place here in the vertex shader. Uniform parameters include the transformation matrices like the MVP, Model and inverse Model transformations. MVP takes the point from modeling space to normalized device space. M takes it to world space for illumination computation, and its inverse is used to transform the normal vector. As illumination happens here, the shader also needs the material properties and light source parameters, as well as the position of the eye, which is used to determine the viewing direction. As varying inputs, the position and the normal associated with the vertex arrive at input registers 0 and 1, respectively.

First, the point is transformed to normalized device space and written to the gl_Position output register, as almost always.

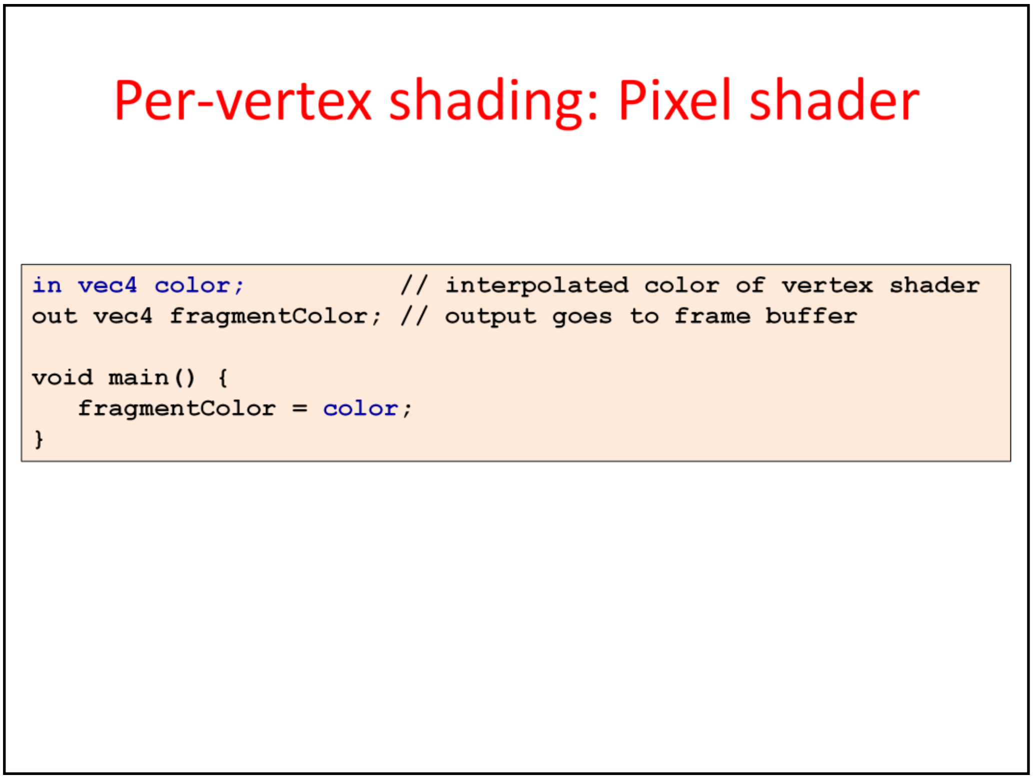
The remaining part of the shader computes the output color. The point is transformed to world space, where it is called as wPos and is in homogeneous coordinates.

Illumination direction points from the shaded point wPos to light source position wLiPos. If they were in Cartesian coordiantes, the difference of the two vectors should be obtained. However, they are in homogeneous coordinates and can even be ideal points being at infinity. If they are not ideal points, homogeneous division would convert homogeneous coordinates to Cartesian coordinates, so the difference can be calculated. As the difference is normalized later on, we can multiply the two vectors by a scalar, the resulting L would not change. Let this scalar be the product of the fourth homogeneous coordinates wLiPos.w * wPos.w. The advantage of this, that the resulting formula would be correct even if either the light source or the shaded point is at infinity. Thus, we can handle point light sources and directional sources in the same way, just directional sources must be placed at ideal points.

Vector V is the viewing direction, which is obtained as the difference of position vectors wEye and the position of the point calculated with homogeneous division.

The normal vector is transformed to world. However, while points are transformed as multiplying their row vector with a transformation matrix from the right, normal vectors are column vectors that are transformed with the inverse matrix being on the left.

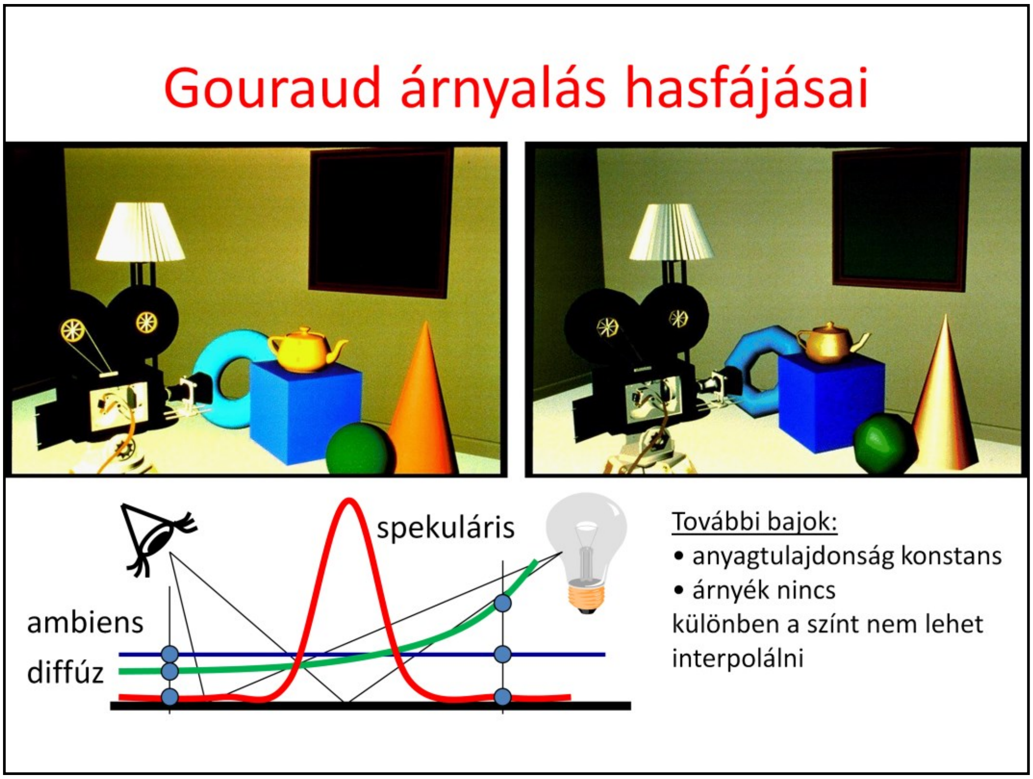
Halfway vector H is computed as adding the unit vectors L and V. The geometry factor is the cosine of the angle between vectors N and L. If its is negative, it is replaced by zero. The Phong-Blinn specular reflection needs the cosine of the angle between the surface normal and the halfway vector, which is obtained by a dot product. Finally, ambient + diffuse + specular reflection model formula is evaluated.



Per-vertex shading: Pixel shader

```
in vec4 color;           // interpolated color of vertex shader
out vec4 fragmentColor; // output goes to frame buffer

void main() {
   fragmentColor = color;
}
```

The fixed function elements interpolate the color variable for internal pixels, which is written into the frame buffer without any modification in case of per-vertex shading.



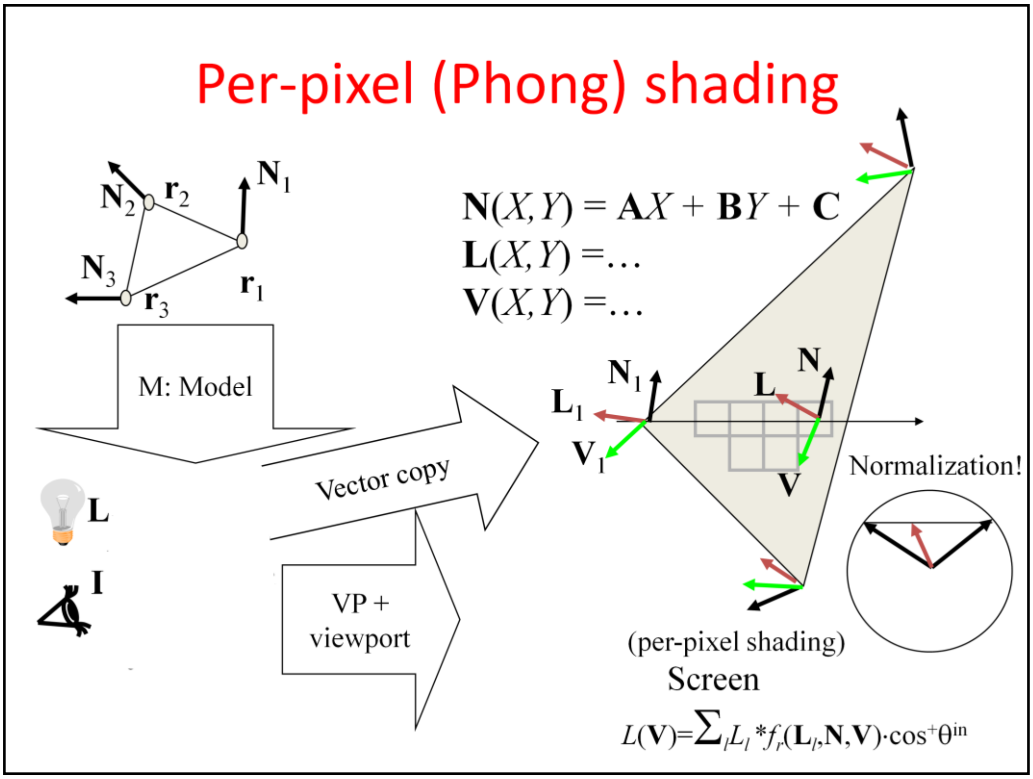
Gouraud árnyalás hasfájásai

<u>További bajok:</u>

- anyagtulajdonság konstans
- árnyék nincs

különben a színt nem lehet interpolálni

Gouraud shading produces satisfactory results if the surfaces are finely tessellated and not strongly glossy or specular. If the triangles are large and the material is highly specular, the highlights will be strangely deformed revealing the underlying triangular approximation. The problem is that glossy reflection is strongly non-linear, which cannot be well represented by linear interpolation.



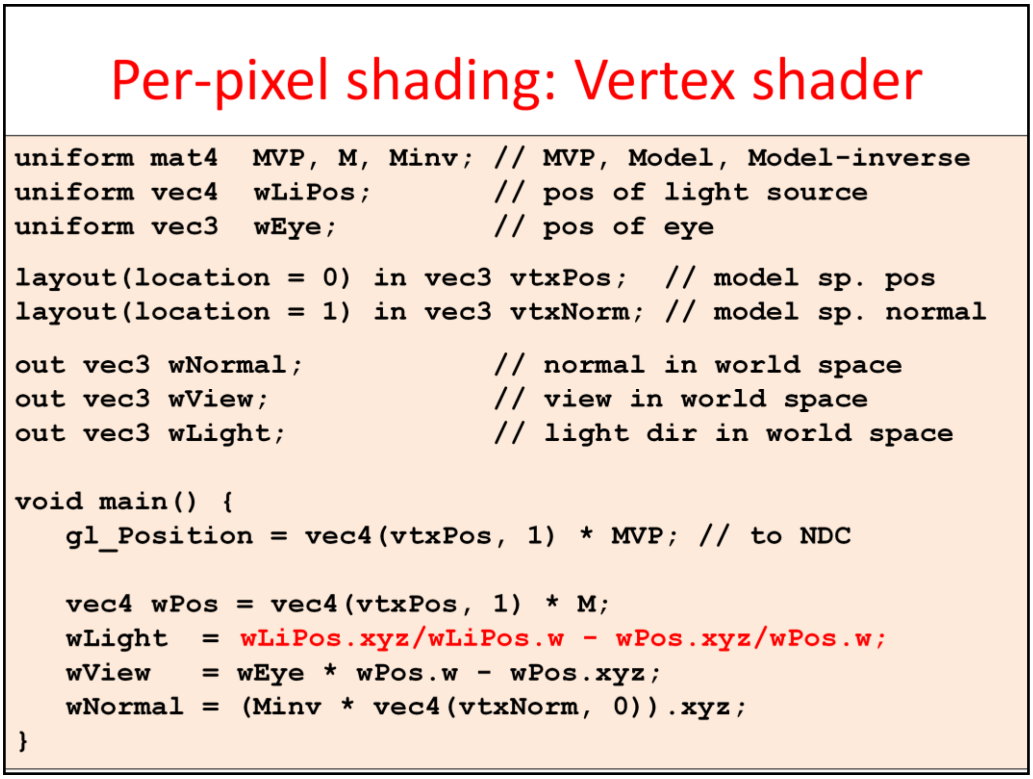
Per-pixel (Phong) shading

$$\mathbf{N}(X,Y) = \mathbf{A}X + \mathbf{B}Y + \mathbf{C}$$
$$\mathbf{L}(X,Y) = \ldots$$
$$\mathbf{V}(X,Y) = \ldots$$

$$L(\mathbf{V}) = \sum_l L_l * f_r(\mathbf{L}_l, \mathbf{N}, \mathbf{V}) \cdot \cos^+\theta^{in}$$

The solution to this problem is Phong shading, which interpolates only those variables which are smoothly changing inside a triangle. Phong shading interpolates the vectors (normal, view, and illumination) and evaluates the radiance from the interpolated vectors at each pixel.

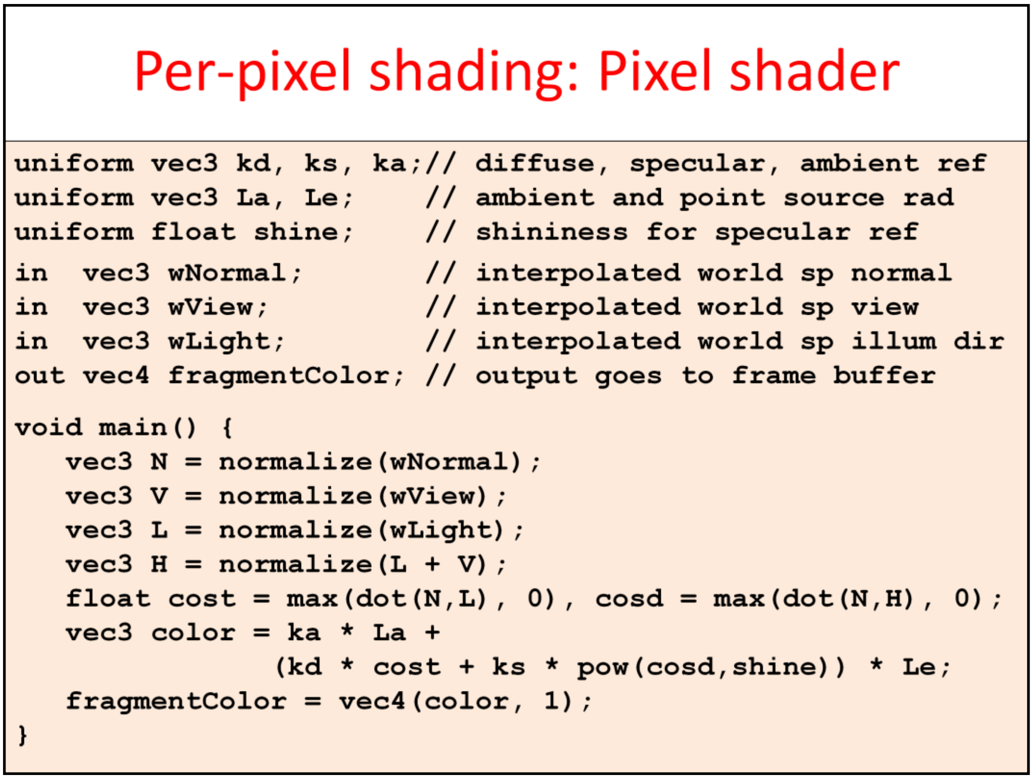
The triangle vertices and their shading normals are transformed to world space (or camera space). Here, illumination and view directions are obtained for each vertex. The vertices are transformed further to screen space, and the computed normal, illumination and view directions will follow them, but without any transformation (they still represent world (or camera) space directions). These vectors are interpolated inside the triangle and for every pixel, where the radiance is obtained with normalizing the vectors and evaluating of the illumination formula for rough surfaces.



Per-pixel shading: Vertex shader

```
uniform mat4  MVP, M, Minv; // MVP, Model, Model-inverse
uniform vec4  wLiPos;       // pos of light source
uniform vec3  wEye;         // pos of eye

layout(location = 0) in vec3 vtxPos;  // model sp. pos
layout(location = 1) in vec3 vtxNorm; // model sp. normal

out vec3 wNormal;           // normal in world space
out vec3 wView;             // view in world space
out vec3 wLight;            // light dir in world space

void main() {
   gl_Position = vec4(vtxPos, 1) * MVP; // to NDC

   vec4 wPos = vec4(vtxPos, 1) * M;
   wLight  = wLiPos.xyz/wLiPos.w - wPos.xyz/wPos.w;
   wView   = wEye * wPos.w - wPos.xyz;
   wNormal = (Minv * vec4(vtxNorm, 0)).xyz;
}
```

In Phong or per-pixel shading, the vertex shader prepares the vectors needed by the illumination formula and outputs them in output registers wNormal, wView and wLight, in addition to the usual task of transforming input point vtxPos from modeling space to normalized device space with the MVP transformation. Preparation includes the transformation of the input point to world coordinates and here obtaining the illumination and viewing directions as the differences of the 3D vectors of the point and light position and eye position, respectively. The output variables are interpolated for every pixel covered by this triangle.

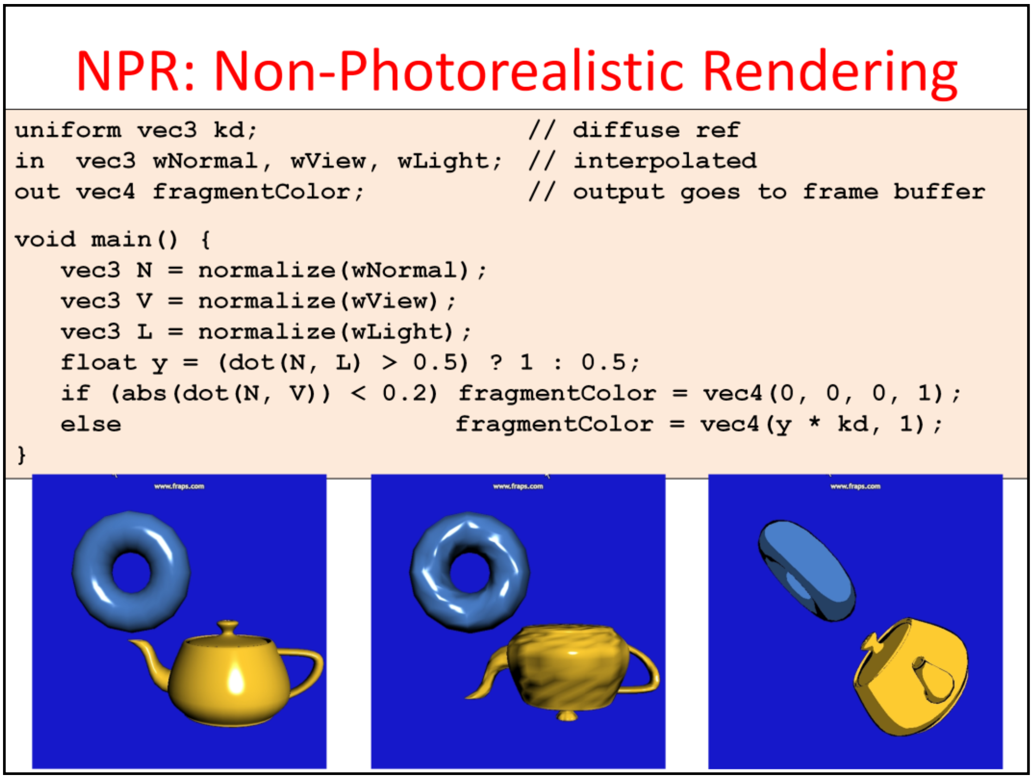


Per-pixel shading: Pixel shader

```
uniform vec3 kd, ks, ka;// diffuse, specular, ambient ref
uniform vec3 La, Le;    // ambient and point source rad
uniform float shine;    // shininess for specular ref

in  vec3 wNormal;       // interpolated world sp normal
in  vec3 wView;         // interpolated world sp view
in  vec3 wLight;        // interpolated world sp illum dir
out vec4 fragmentColor; // output goes to frame buffer

void main() {
   vec3 N = normalize(wNormal);
   vec3 V = normalize(wView);
   vec3 L = normalize(wLight);
   vec3 H = normalize(L + V);
   float cost = max(dot(N,L), 0), cosd = max(dot(N,H), 0);
   vec3 color = ka * La +
               (kd * cost + ks * pow(cosd,shine)) * Le;
   fragmentColor = vec4(color, 1);
}
```

The pixel shader gets the interpolated vectors and also material and light source parameters in uniform variables.

First the normal, view and light vectors are normalized to obtain N, V, L. Then, the halfway vector H is computed, and the ambient + diffuse + Phong-Blinn specular illumination formula is used to compute the reflected color, which is output to the frame buffer.

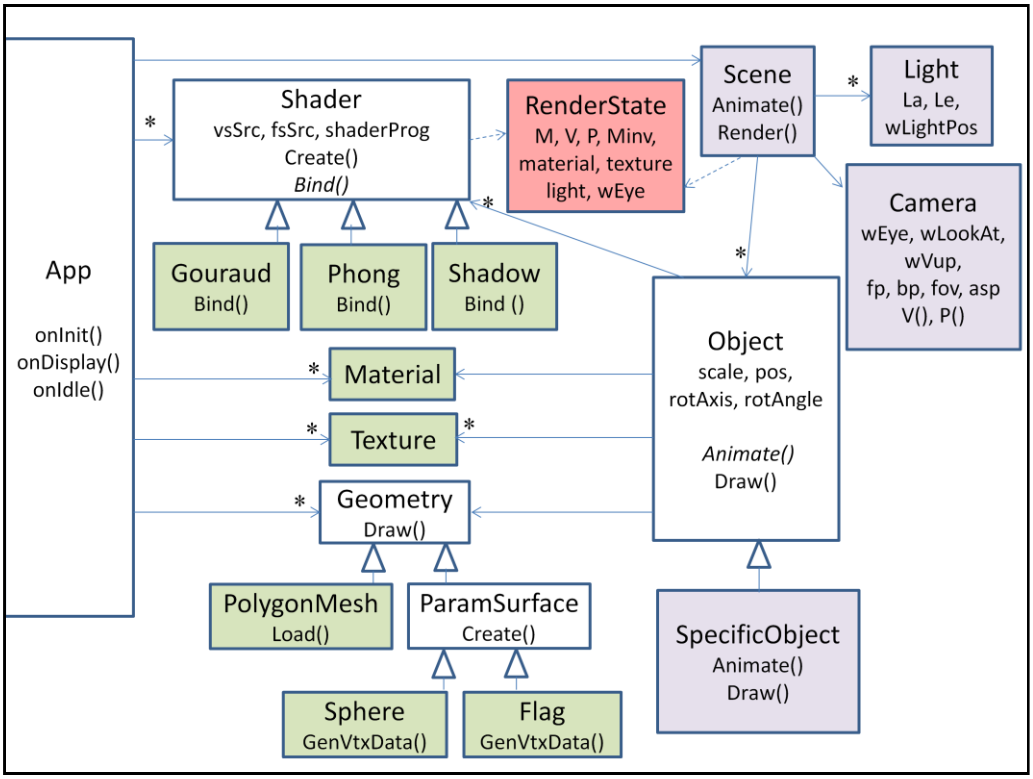


NPR: Non-Photorealistic Rendering

```
uniform vec3 kd;                   // diffuse ref
in  vec3 wNormal, wView, wLight;  // interpolated
out vec4 fragmentColor;           // output goes to frame buffer

void main() {
   vec3 N = normalize(wNormal);
   vec3 V = normalize(wView);
   vec3 L = normalize(wLight);
   float y = (dot(N, L) > 0.5) ? 1 : 0.5;
   if (abs(dot(N, V)) < 0.2) fragmentColor = vec4(0, 0, 0, 1);
   else                      fragmentColor = vec4(y * kd, 1);
}
```

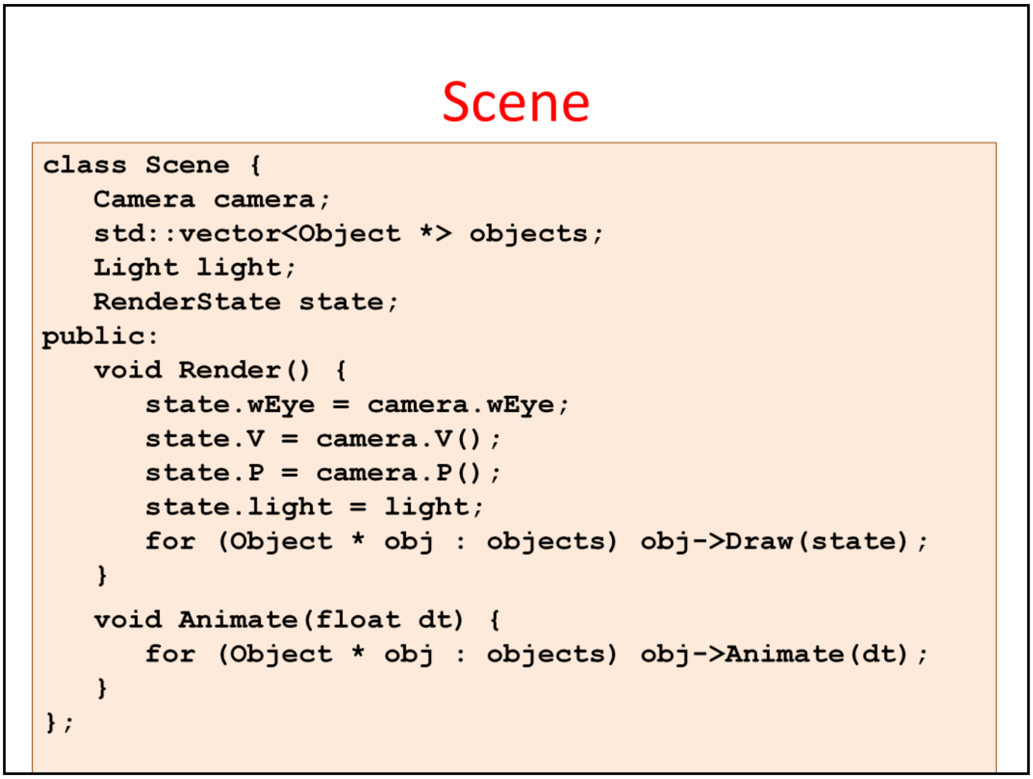
www.fraps.com
www.fraps.com
www.fraps.com

NPR or non-photorealistic rendering means the simulation of the artistic process of painting. As the painter has limited number of paints, colors are quantized, and the silhouette are drawn in black. The silhouette is identified where the surface normal is approximately orthogonal to the viewing direction, i.e. their dot products is close to zero.



This is the UML class diagram of a simple 3D engine. Our virtual world is an object of type Scene, which can be animated and rendered. A scene has arbitrary number of Light sources, a single Camera and heterogeneous collection of Objects. An Object has animation parameters determining scaling, translation and rotation and can be animated by setting these transformation parameters and can also be drawn with the help of OpenGL. Of course, the specific way of drawing and possibly animation is object type dependent, so these are virtual functions that the can re-defined by inheritance. For drawing, an object needs Material properties, Texture, Geometry in the form of a VAO and a Shader that specifies the programs processing vertices and pixels associated with the object. As many objects may share the same shader, material, geometry or texture, objects do not contain these, but have an association with them. It means that the Application creates a collection of shaders, materials, etc. and each object can pick from them according to its taste.

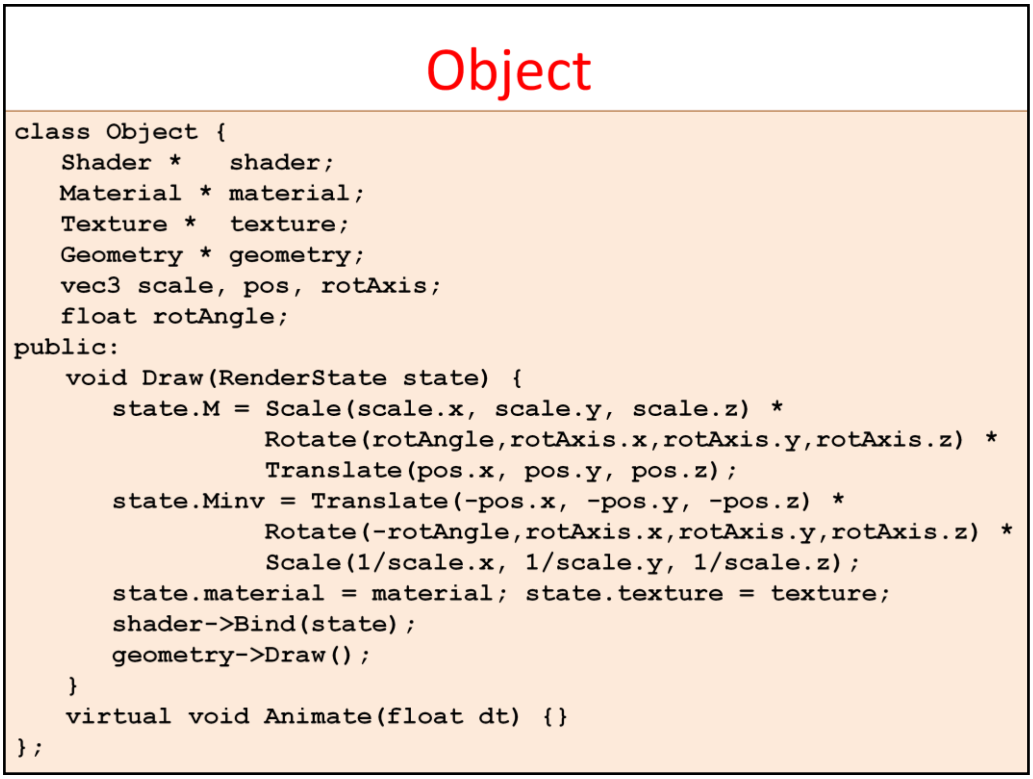
The relation of the Shader and the Object needs special care. Object properties apart from the vertex data should show up as uniform variables. The question is whose responsibility should be this copy. If it is the Shader's, the Object structure becomes rigid. If it is the Object's, the Shader cannot be changed without the modification of the Object. Thus, we introduce a communicator object, called RenderState. Object fills up the RenderState object as it wishes, and Shader fills up its uniform variables based on the content of the RenderState.



Scene

```
class Scene {
   Camera camera;
   std::vector<Object *> objects;
   Light light;
   RenderState state;
public:
   void Render() {
      state.wEye = camera.wEye;
      state.V = camera.V();
      state.P = camera.P();
      state.light = light;
      for (Object * obj : objects) obj->Draw(state);
   }
   void Animate(float dt) {
      for (Object * obj : objects) obj->Animate(dt);
   }
};
```

A Scene has a Camera, heterogeneous collection of objects, a light source, and it also uses a state of type RenderState. Rendering means the initialition of the state according to the camera and lights, and then calling each object to draw itself based on the initialized state.

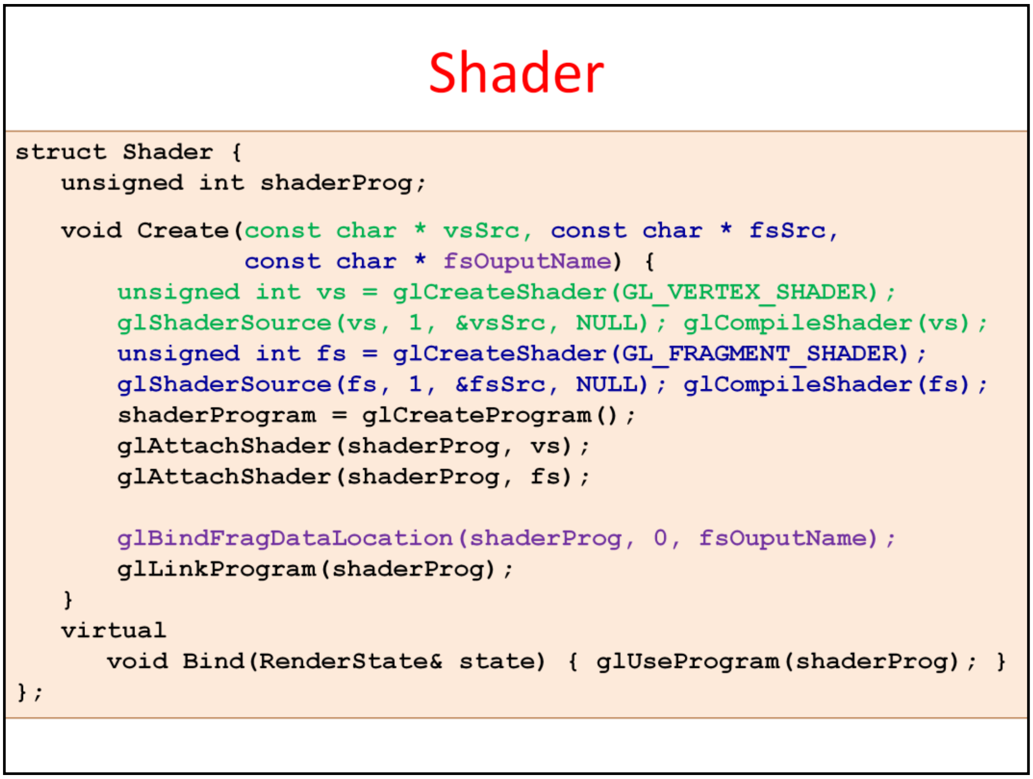


Object

```
class Object {
   Shader *   shader;
   Material * material;
   Texture *  texture;
   Geometry * geometry;
   vec3 scale, pos, rotAxis;
   float rotAngle;
public:
   void Draw(RenderState state) {
      state.M = Scale(scale.x, scale.y, scale.z) *
                Rotate(rotAngle,rotAxis.x,rotAxis.y,rotAxis.z) *
                Translate(pos.x, pos.y, pos.z);
      state.Minv = Translate(-pos.x, -pos.y, -pos.z) *
                Rotate(-rotAngle,rotAxis.x,rotAxis.y,rotAxis.z) *
                Scale(1/scale.x, 1/scale.y, 1/scale.z);
      state.material = material; state.texture = texture;
      shader->Bind(state);
      geometry->Draw();
   }
   virtual void Animate(float dt) {}
};
```

Object is associated with a shader, a material, a texture, a geometry, and has parameters determining the transformation, i.e. the animation.

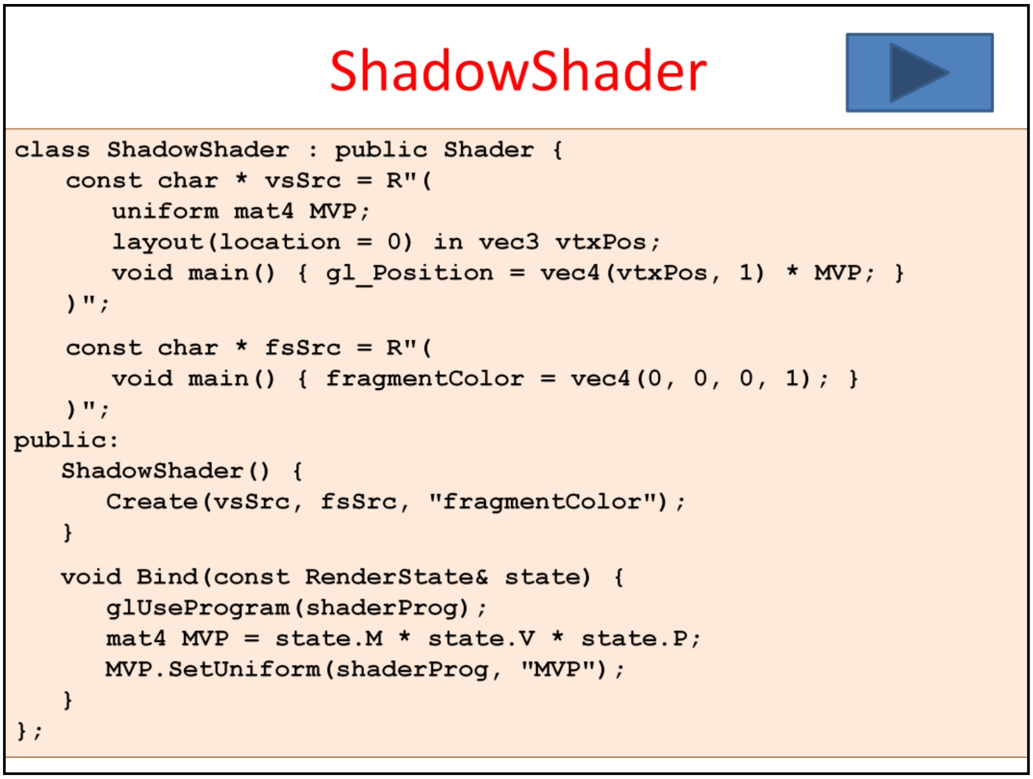
During drawing, modeling transformation matrix M is set based on the transformation parameters, and also its inverse is computed. Material and texture are also set. The shader’s bind activates this data in the uniform variables of the shader. The geometry’s draw triggers the rendering pipeline to process the vertices in the vbos of the vao of the current object.



Shader

```
struct Shader {
   unsigned int shaderProg;

   void Create(const char * vsSrc, const char * fsSrc,
               const char * fsOuputName) {
       unsigned int vs = glCreateShader(GL_VERTEX_SHADER);
       glShaderSource(vs, 1, &vsSrc, NULL); glCompileShader(vs);
       unsigned int fs = glCreateShader(GL_FRAGMENT_SHADER);
       glShaderSource(fs, 1, &fsSrc, NULL); glCompileShader(fs);
       shaderProgram = glCreateProgram();
       glAttachShader(shaderProg, vs);
       glAttachShader(shaderProg, fs);

       glBindFragDataLocation(shaderProg, 0, fsOuputName);
       glLinkProgram(shaderProg);
   }
   virtual
      void Bind(RenderState& state) { glUseProgram(shaderProg); }
};
```

A Shader gets the source codes of the vertex and fragment shader programs and programs the shader processors with them.



ShadowShader

▶

```
class ShadowShader : public Shader {
    const char * vsSrc = R"(
        uniform mat4 MVP;
        layout(location = 0) in vec3 vtxPos;
        void main() { gl_Position = vec4(vtxPos, 1) * MVP; }
    )";

    const char * fsSrc = R"(
        void main() { fragmentColor = vec4(0, 0, 0, 1); }
    )";
public:
    ShadowShader() {
        Create(vsSrc, fsSrc, "fragmentColor");
    }

    void Bind(const RenderState& state) {
        glUseProgram(shaderProg);
        mat4 MVP = state.M * state.V * state.P;
        MVP.SetUniform(shaderProg, "MVP");
    }
};
```

This is a simple example shader drawing everything in black. The vertex shader transforms the points with MVP and the fragment shader sets all pixels to black that are covered by the rendered geometry. As this shader has just a single uniform variable, the MVP matrix, this is copied in the Bind function.

Such shaders can be used to render projected shadows.