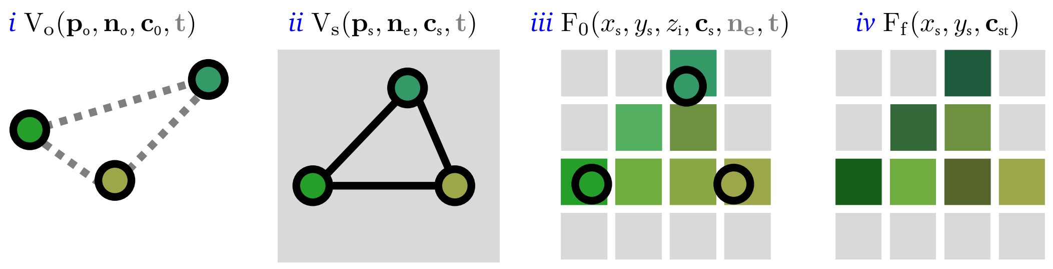
Programmable Rasterization

ECSE 546 / COMP 599 – Scribe Note

Adrian Koretski 260587339

Anjun Hu 260765961

1.1 Programmable Rasterization



i $\mathrm{V}_0(\mathbf{p}_o, \mathbf{n}_o, \mathbf{c}_0, \mathbf{t})$
ii $\mathrm{V}_s(\mathbf{p}_s, \mathbf{n}_e, \mathbf{c}_s, \mathbf{t})$
iii $\mathrm{F}_0(x_s, y_s, z_i, \mathbf{c}_s, \mathbf{n}_e, \mathbf{t})$
iv $\mathrm{F}_f(x_s, y_s, \mathbf{c}_{st})$

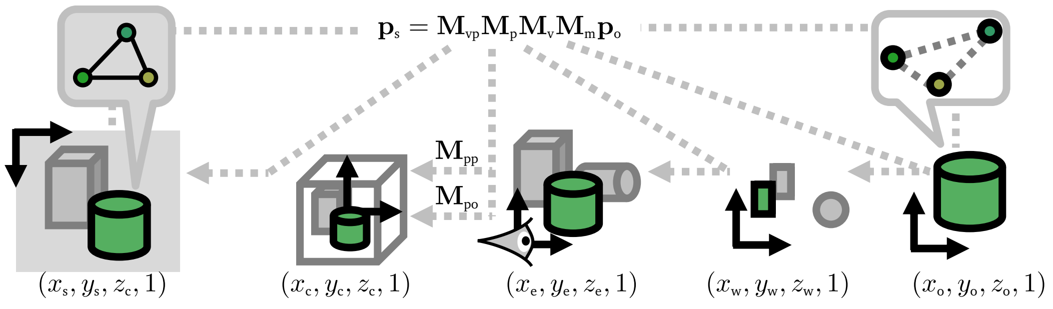
The programmable rasterization pipeline can be decomposed into four stages, namely (*i*) vertex processing, (*ii*) clipping and rasterization, (*iii*) fragment shading and (*iv*) output display of a shaded and textured image. Input variables at each stage are shown, some of which are optional.

Overview Rendering, or image synthesis, is the generation of image pixel data through interpretation of information about a scene. Two problems situate at the very foundation of this task, namely, visibility and shading. More colloquially, to create an image of a scene, we need to know whether a point in the scene is observable to the viewer, and if so, what color would be seen.

Programmable rasterization is a widely employed pipeline to solve these two problems. It starts from the scene geometry and solves for on-screen appearance with heavy reliance on projections and transformations. It is said to be an "object-centric" approach that is in contrast to the "view-centric" ray-tracing technique where one initiates a ray from the image screen and seeks for intersections between the ray and the scene.

The pipeline takes as input a scene with 3D objects. Objects are meshes of *triangular primitives*[1], or equivalently, sets of *vertices* stored in the *object space* with various attributes as depicted above. Each primitive is projected onto the screen and rasterized into *fragments* that entail discrete positions in the screen space with interpolated depths. The aforementioned visibility and shading problems are then resolved on a per-fragment basis.

Basic Idea Stage *i* of the pipeline encompasses the processing of vertex attributes. Vertex positions $\mathbf{p}_o$ in an object's local space are fed into the pipeline along with other information, which may include surface normal directions in the object space $\mathbf{n}_o$, unshaded color $\mathbf{c}_0$ and texture $\mathbf{t}$. As depicted below, vertices are mapped onto the image plane through a sequence of vector[2] transformations described as matrix multiplications as shown below. Details will be revisited.



In the vertex processing stage *i*, vertex position vectors are sequentially mapped from object space to world space, eye space, canonical view volume and eventually the screen space through matrix multiplications. One could represent this chain of transformation as one single 4 by 4 matrix.

[1]The same concepts can be extended to quadrilateral and other polygonal primitives.
[2]Spatial vectors are stored as 4-tuples. The additional dimension indicates whether it is a direction (0) or a position (1).

With the transformed set of vertices $\mathbf{p}_{\text{s}}$, we may proceed to stage *ii* where clipping and rasterization take place. We aim to map each primitive to the set of pixels it would cover on the screen. This can be done in various ways, among which we will be presenting the *edge functions* and the *Barycentric coordinates*. Other geometric and textural attributes are interpolated.

Stage *iii* centers around fragment shading. *Visibility tests* are employed to omit unseen fragments that does not require shading. They could be based on depth, stencil or surface normal direction. Shading, or color assignment, could be done in a per-triangle, per-vertex or per-pixel manner depending on the shading scheme. One could also overlay textures towards the end of this stage.

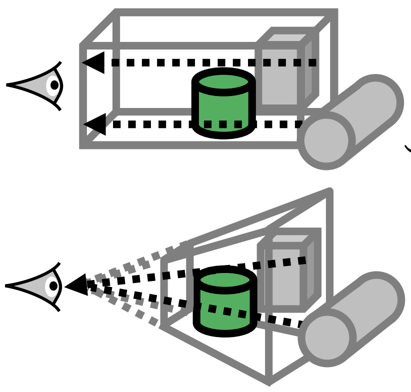
Finally, shaded color of visible fragments are written into the framebuffer with optional gamma correction. By the end of stage *iv*, the rendered image is ready for display.

Algorithm Details An input vertex may have a descriptor in the form of $\text{V}_{\text{O}}(\mathbf{p}_{\text{o}}, \mathbf{n}_{\text{o}}, \mathbf{c}_{0}, \mathbf{t})$, namely, its position and normal direction in object space as 4D vectors, an unshaded color as a 3D vector and a texture as a 2D vector. Left multiplication by the *model* matrix $\mathbf{M}_{\text{m}}$ first transforms $\mathbf{p}_{\text{o}}$ into the world space. Then with the *viewing* matrix $\mathbf{M}_{\text{v}}$ on the world space position, $\mathbf{p}_{\text{w}}$ we arrive at the eye space where a fictitious viewer situates at the origin. The normal directions in the eye space can be obtained via $\mathbf{n}_{\text{e}} = ((\mathbf{M}_{\text{v}}\mathbf{M}_{\text{m}})^{-1})^{\text{T}}\mathbf{n}_{\text{o}}$.

For certain shading schemes, the per-triangle shaded color (as in flat shading) or the per-vertex shaded color (as in Gouraud shading) $\mathbf{c}_{\text{s}}$ is computed with the input unshaded color $\mathbf{c}_{0}$ and the normal direction of a triangle or a vertex. Alternatively, one could adopt the Phong shading scheme and shade with per-pixel normal, in which case the computation of color is delayed to stage *iii* and $\mathbf{c}_{0}$ can be passed forward as-is. At this point, we are ready to enter the screen space.

$$\underbrace{\begin{bmatrix} x_s \\ y_s \\ z_c \\ 1 \end{bmatrix}}_{\mathbf{p}_{\text{s}}} = \underbrace{\begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathbf{M}_{\text{vp}}} \mathbf{M}_{\text{p}} \underbrace{\begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1}}_{\mathbf{M}_{\text{v}}} \mathbf{M}_{\text{m}} \underbrace{\begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}}_{\mathbf{p}} \tag{1}$$

We define a *view frustum* as the volume of interest in the eye space that will be projected onto the screen. The volume can be taken as a box with parallel rays in an *orthographic* case. Otherwise, it is said to be *perspective* and would resemble a trapezoidal prism in which viewing rays converge to the viewer. This volume is then mapped into a cubic canonical view volume $[-1, 1]^3$ using the corresponding projection matrix, $\mathbf{M}_{\text{po}}$ or $\mathbf{M}_{\text{pp}}$.



$$\underbrace{\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{l+r}{r-l} \\ 0 & \frac{2n}{t-b} & 0 & -\frac{b+t}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{f+n}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathbf{M}_{\text{po}}}$$

$$\underbrace{\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{l-r} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{b-t} & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\mathbf{M}_{\text{pp}}}$$

Projection matrices *i*: Unlike the affine **orthographic** transformation, the **perspective** transformation accounts for field of view and thus, does not retain equidistance between points or parallelism between lines.

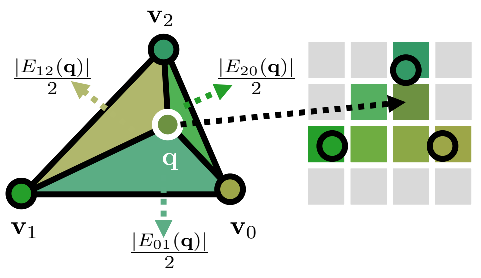
Finally, the leftmost matrix $\mathbf{M}_{\text{vp}}$ projects the view frustum onto the viewport and marks the completion of stage *i*. Note that we still have a continuous representation of the primitives.

In stage *ii*, each primitive is mapped to a set of discrete screen elements, the *fragments*[3]. We first make pairwise connections between the triangle vertices $\{\mathbf{v}_i \mid \forall i \in \{0,1,2\}\}$ to obtain three lines, each of which partitions the image plane into two regions. Then the *edge function* $E_{ij}(\mathbf{q})$ is applied to each fragment location $\mathbf{q}$ to check if it is "contained" in a triangle, that is, if it situates in positive half plane with respect to all three triangle edges[4]. This is equivalent to taking the cross product of two vectors $\mathbf{q}-\mathbf{v}_\text{i}$ and $\mathbf{v}_\text{j}-\mathbf{v}_\text{i}$ for each edge (i,j), and then examine the direction. The edge function is essentially twice the *directed* area of the "sub-triangle" defined by the two vertices $\mathbf{v}_i, \mathbf{v}_j$ and the fragment $\mathbf{q}$. It is aligned with the normal direction of the fragment.[5]

$$E_{ij}(\mathbf{q}) = (x_\mathbf{q} - x_{\mathbf{v}_i})(y_{\mathbf{v}_j} - y_{\mathbf{v}_i}) - (y_\mathbf{q} - y_{\mathbf{v}_i})(x_{\mathbf{v}_j} - x_{\mathbf{v}_i}) = (\mathbf{q} - \mathbf{v}_i) \times (\mathbf{v}_j - \mathbf{v}_i) \tag{2}$$

Now each triangle is mapped to a set of fragments $\text{F}_0(x_\text{s}, y_\text{s}, z_\text{i}, \mathbf{c}_1, \mathbf{t})$. However, color and other information is missing for all but the three fragments that coincide with the vertices. We would like to infer the missing properties of each fragment from the three surrounding vertices. A 3-tuple called the *Barycentric coordinates* serves this very purpose. It allows for interpolation of fragment attributes by representing each fragment as a weighed geometric mean of the three vertices:

$$B(\mathbf{v}_i, \mathbf{q}) = \frac{Area(\mathbf{v}_j, \mathbf{v}_k, \mathbf{q})}{Area(\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k)} = \frac{1}{2}\frac{E_{jk}(\mathbf{q})}{Area(\Delta)} \tag{3}$$

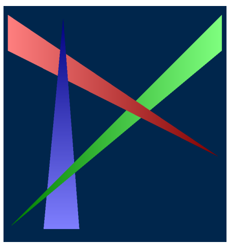


$\mathbf{v}_2$
$\frac{|E_{12}(\mathbf{q})|}{2}$
$\frac{|E_{20}(\mathbf{q})|}{2}$
$\mathbf{q}$
$\mathbf{v}_1$
$\mathbf{v}_0$
$\frac{|E_{01}(\mathbf{q})|}{2}$

Barycentric Coordinates *ii*: Properties of an "interior" fragment $\mathbf{q}$ is interpolated as a weighed average of three triangle vertices. The weight of each vertex is proportional to the area of its corresponding sub-triangle. Note the clockwise winding convention.

Remarks: Extra care should be taken when interpolating depth z_i with a perspective view frustum. Perspective transformations are not affine. Therefore, linear interpolation of depth in the post-projection space is not equivalent to linear interpolation of z_w in the world space. Our goal is to allocate bins that are equally spaced in the *post-projection space* for best visual outcomes, which can be done by interpolating the reciprocal of depth.

In stage *iii*, the fragment shading process starts from solving the *visibility problem*. To omit primitive that are invisible due to occlusion, one may intuitively consider the naive painter's algorithm, that is, drawing the scene from back to front and overwriting along the way. However, this approach fails when "entangled" geometry exists in a scene. Instead, we can maintain the depth on a per-fragment basis with the *z-buffer* technique. For each position $[x, y]$ on the screen, an extra channel is allocated to keep track of the closest depth so far. For each fragment, we compare its depth to the closest depth at its designated screen position, and overwrite the previous result only if the new fragment is closer, that is, have a smaller z-value.



Z-buffer : Painter's algorithm fails in this case

If we adhere to the Phong shading scheme, we may now shade with an interpolated fragment color and a per-pixel normal direction. Optionally, textures can be overlaid. We have now obtained a color for each pixel, which would be written into the framebuffer at stage *iv* and be displayed on the screen.

[3]We assume that each pixel corresponds to one fragment in this report. One may opt for multi-sample anti-aliasing, in which case multiple fragments are used to shade one single pixel.
[4]Depending on the winding of the pixels, alternative conventions (negative as "in" and positive as "out") can be chosen.
[5]There are many alternative ways to check for containment, which are usually more efficient. We use the edge functions for simplicity and demonstration purposes.

Implementation Details For simplicity, we assume that one fragment maps to one screen pixel and multi-sampling anti-aliasing is not applied. Also, it is recommended that one organize the scene in such a way that $\mathbf{M}_{\mathrm{m}}$ is the identity matrix.

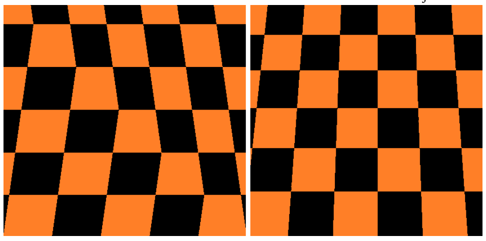
The pseudocode below assumes that `Vertex` objects are formulated as $\mathrm{V}(\mathbf{p}(x, y, z), \mathbf{n}, \mathbf{c}, \mathbf{t})$ and `Fragment` objects are formulated as $\mathrm{F}(\mathbf{p}(x_{\mathrm{s}}, y_{\mathrm{s}}, z_{\mathrm{i}}), \mathbf{c}_{\mathrm{s}}, \mathbf{n}_{\mathbf{e}}, \mathbf{t})$. It's also assumed that we have access to utility functions to calculate `Barycentric` coordinates based on edge functions, which are also used to judge whether or not a `Fragment` `isContained` in a `Prim`itive.

```
1  void Rasterize (Scene scene, Screen screen){
2
3   // Loop over all objects, all primitives
4   for (Object obj : scene.objects){
5    for (Prim triangle : obj){
6
7     /* Stage i: vertex transformation and projection */
8     for (Vertex vo : triangle){
9      Vertex ve, vs;
10     // transform from object space to eye space
11     ve.p = Mv * Mm * vo.p;
12     ve.n = transpose(inv(Mv * Mm)) * vo.n;
13     // Shading with per-vertex normal
14     if (shadeGouraud) { shade(ve); }
15    // Shading with per-face normal
16    if (shadeFlat) { shade(triangle); }
17    // transform from eye space to screen space
18    for (vertex : triangle)
19     vs.p = (Mvp * Mp) * ve.p;
20    }
21
22    /* Stage ii: rasterization */
23    for (Fragment frag : screen){
24     // Compute the edge function
25     if isContained(triangle, frag.p.xy);
26      // Compute Barycentric coordinates
27      vec3 w = Barycentric (triangle, frag.p.xy);
28      // Interpolate with Barycentric weights
29      interpolate(w, &frag);
30
31      /* Stage iii: fragment shading */
32      // Z-buffer with interpolated screen depth
33      if (frag.p.z < zbuffer[frag.p.xy]){
34       screen.zbuffer[frag.p.xy] = frag.p.z
35       // Shading with per-pixel normal
36       if (shadePhong) { shade(frag); }
37
38        /* Stage iv: framebuffer writing */
39        screen.framebuffer[frag.xy] = frag.color;}
40    }
41   }
42  }
43 }
```

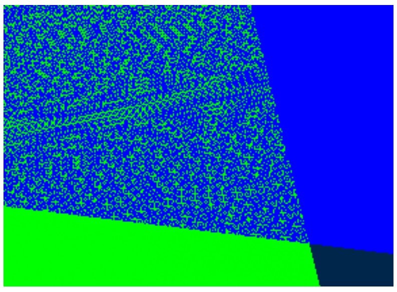
Storing bounding boxes of primitives upon projection is a way to optimize `isContained`. Rather than scanning the entire image plane for every primitive, we only test the pixels inside the bounding box of a primitive for containment (line 25).

Most fragment attributes such as color $\mathbf{c}_1$, normal $\mathbf{n}_{\mathrm{w}}$ and texture $\mathbf{t}$ are interpolated in the world space with respect to Barycentric weights (line 29). This is not the case for depth z_i if we choose to use a perspective view frustum. To avoid erroneous results (left image below), we must ensure linearity in the post-projection screen space. It can be done by interpolating the *reciprocal* of fragment depth in the pre-projection space. Other approaches exist.

$$\frac{1}{z_{\mathbf{q}}} = (1 - B(\mathbf{v}_k, \mathbf{q}))\frac{1}{z_{\mathbf{v}_{\mathrm{i}}}} + B(\mathbf{v}_k, \mathbf{q})\frac{1}{z_{\mathbf{v}_{\mathrm{j}}}}$$



Back-face culling can serve as another visibility test. The *directed* area of each primitive is calculated with the edge functions. Since this vector value is aligned with the primitive's normal direction, it can be used to eliminate fragments whose normal vectors point away from the camera. As a side note, we emphasize that one should keep a consistent winding order of vertices.



Z-fighting may happen when multiple fragments have close values in the z-buffer, which leads to ambiguous results of the visibility test. Enhancing the precision of the z-buffer could relieve this problem. Since precision is distributed between the near and far clipping planes, one should tailor the view frustum to fit the scene without being overly extensive. It is also noted that linear interpolation of depth in the screen space is essential to ensure that we have more precision bins and better accuracy closer to the viewer (line 33).