Lecture 13: Rasterization-based rendering

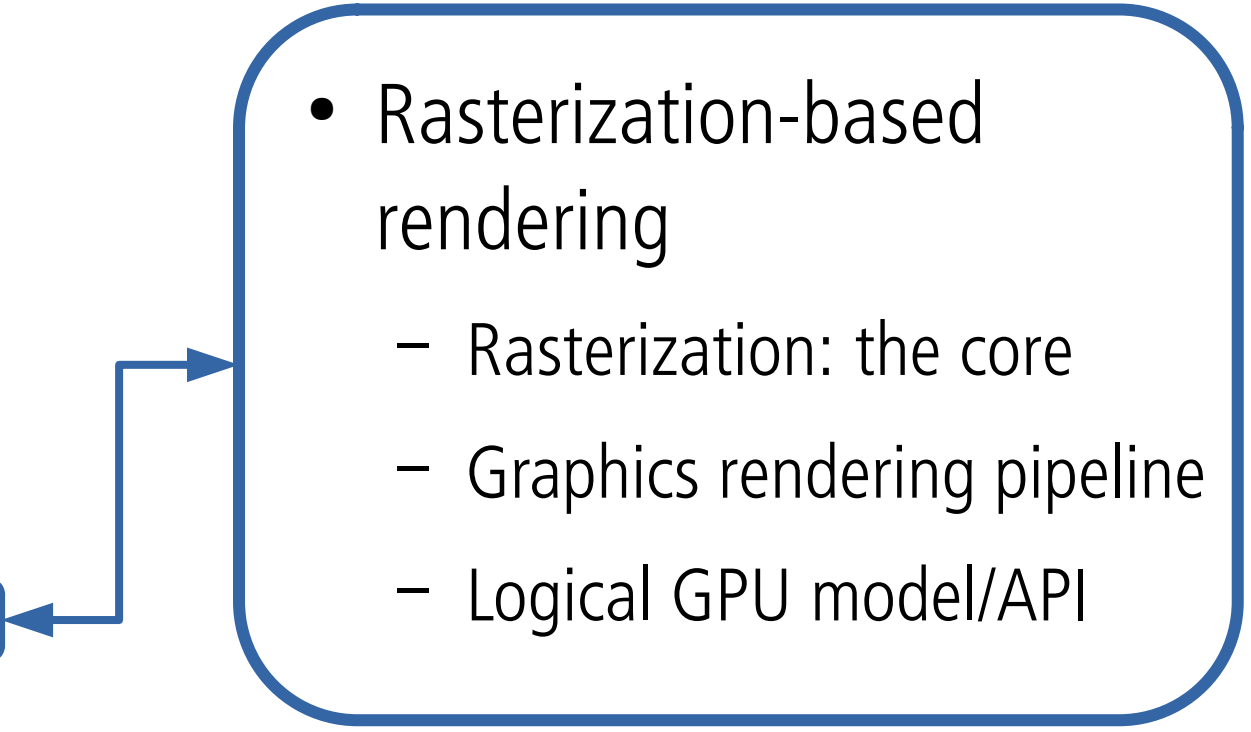
DHBW, Computer Graphics

Lovro Bosnar

22.3.2023.

Syllabus

- 3D scene
  - Object
  - Light
  - Camera
- Rendering
  - Ray-tracing based rendering
  - Rasterization-based rendering
- Image and display

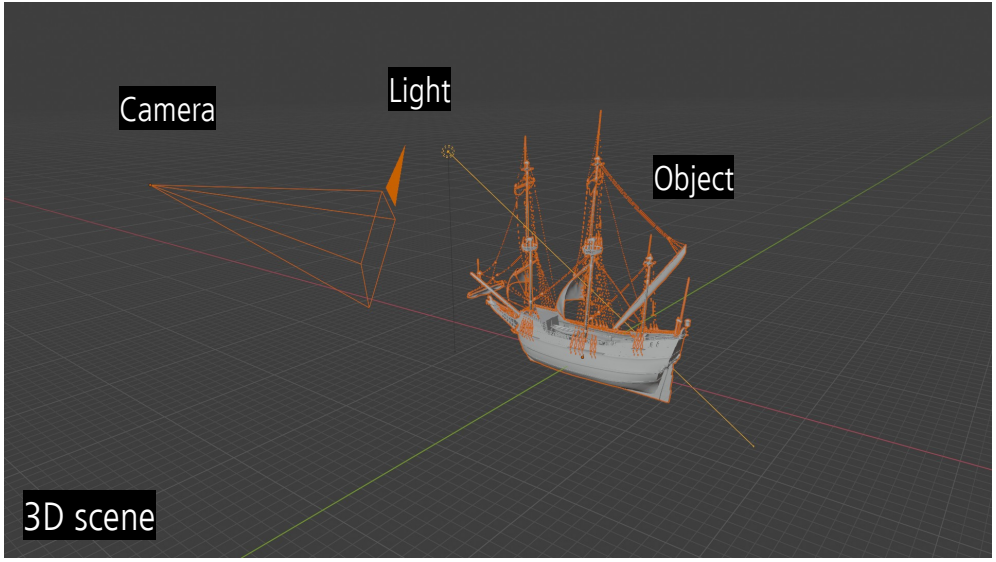


Rasterization

The core for visibility solving

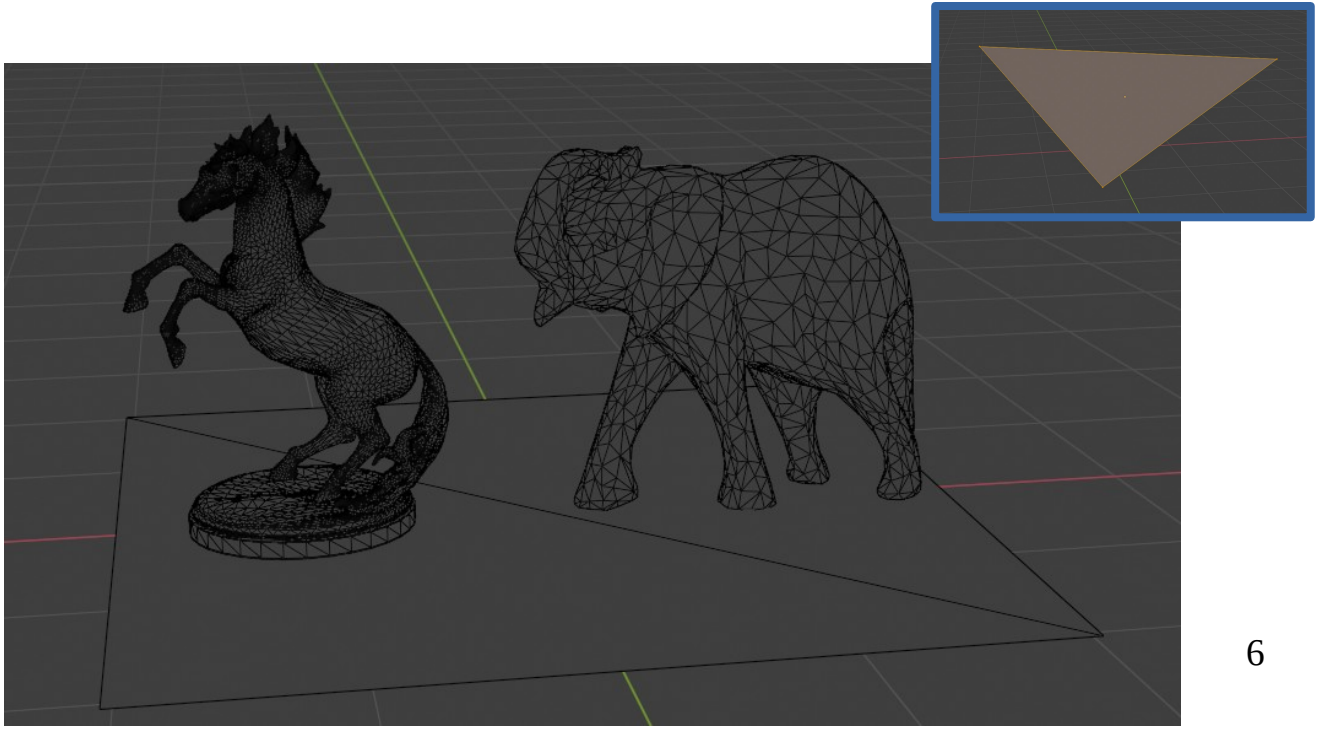
Rendering

- **Rendering: process of creating 2D image from 3D scene**
- Two main **tasks of rendering**:
  - **Visibility solving**: compute objects visible from camera
  - **Shading**: compute colors of visible objects



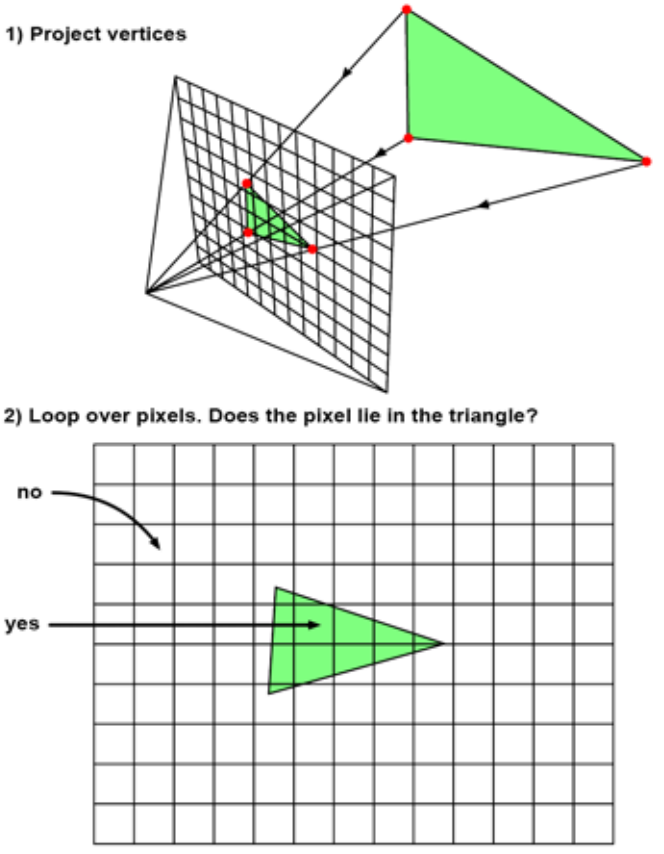
Visibility problem

- **Visibility is geometrical problem using object shape representation** to determine which points in 3D scene are visible to each other
- **Triangulated mesh is shape representation used for efficient visibility solving**
  - Different shape representations can be converted to triangulated mesh using **triangulation**
- **Main approaches for solving visibility:**
  - **Ray-tracing**
  - **Rasterization**



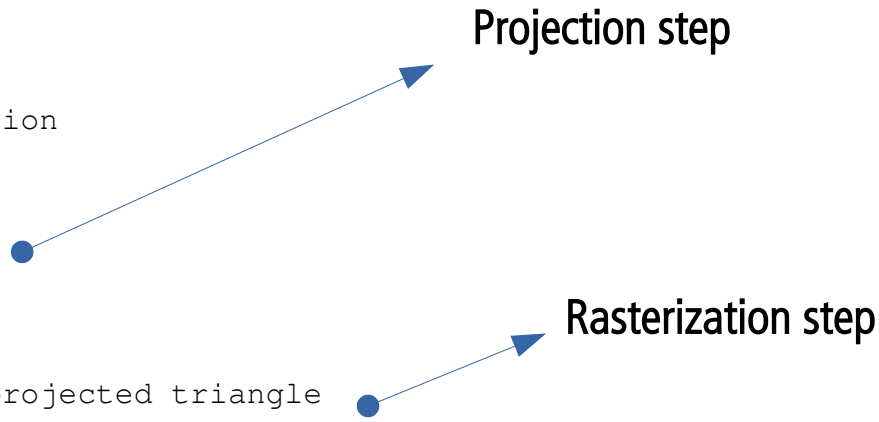
Rasterization: visibility solving

- To **find objects visible from camera, rasterization** can be decomposed in **two main stages**:
  - **Triangle projection**: project 3D vertices of mesh triangles onto image plane (aka screen) using perspective projection matrix
  - **Triangle rasterization**: for each pixel of image plane check if it is contained in projected triangle



Rasterization: algorithm

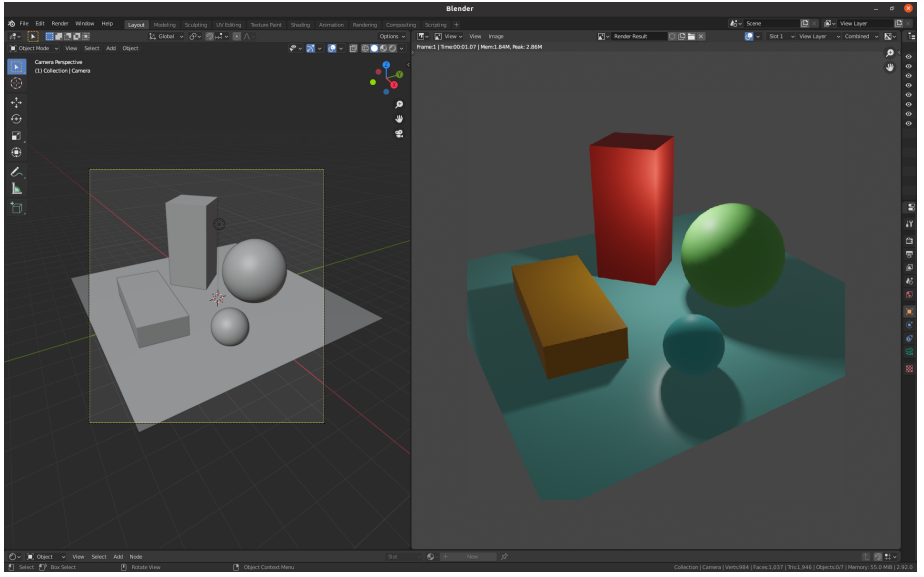
```
for each triangle in the scene:
    // STEP 1: project vertices using perspective projection
    vector v0 = perspectiveProjection(triangle.v0);
    vector v1 = perspectiveProjection(triangle.v1);
    vector v2 = perspectiveProjection(triangle.v2);
    for each pixel in the image:
        // STEP 2: compute if pixel is contained in the projected triangle
        if(pixelCointainedIn2DTriangle(triangle.v0, triangle.v1, triangle.v2,pixel.x, pixel.y):
            image(pixel.x, pixel.y) = triangle.color
```



- **Object centric approach**: first loops over triangles, then loops over image pixels
- Note: **looping over all triangles in 3D scene and all pixels in image is computationally expensive**: further optimizations are introduced (e.g., culling)

Rasterization: image and color buffer

- **Rasterization**: process of **decomposing triangles into pixels → raster image**
- During the process of rasterization, pixel color is stored in a **color buffer – a 2D array of pixels**
- After rasterization, **color buffer contains image** ready for display

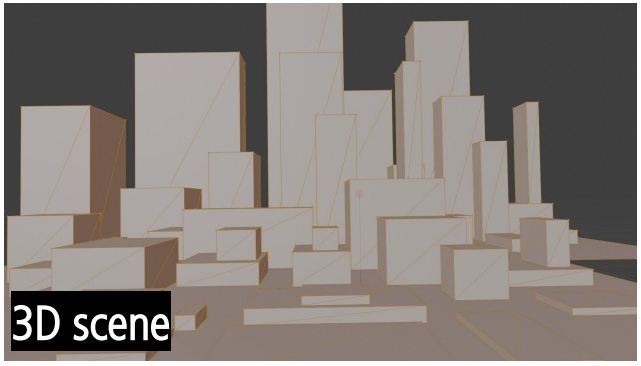


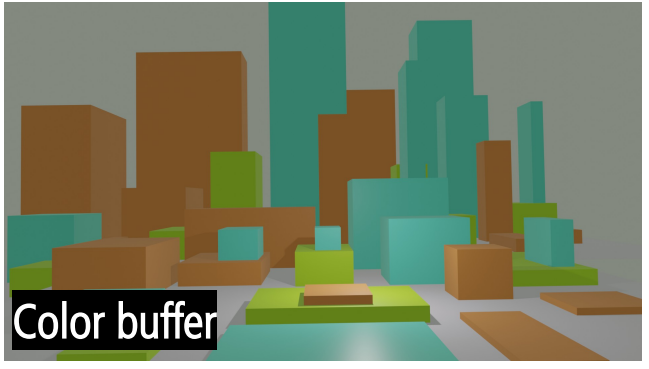
3D scene left. Color buffer, result of the rasterization-based rendering (right)

- **Before rendering**, color-buffer is created with all pixels set to black color
- **During rendering**, when triangles are rasterized, for each pixel that overlaps triangle store that triangle color in color-buffer at that pixel location
- **After rendering**, when all triangles are rasterized and pixels are processed, the color-buffer will contain the final image of the scene

Rasterization: buffers

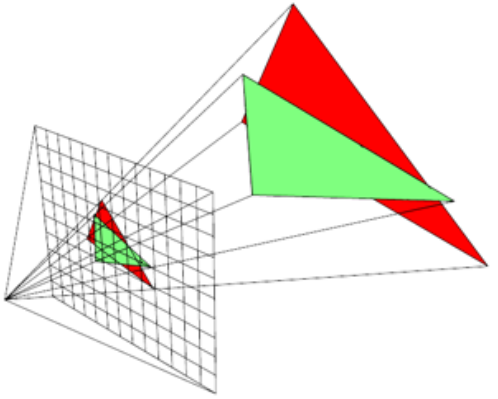
- **During rasterization-based rendering, various pixel information is stored in buffers**
  - **Buffer**: 2D array of pixels which has same size as final image
- **Framebuffer** generally consists of all buffers on a system
  - **Multiple different buffers can be created** during rasterization-based rendering for achieving various effects
- **Common buffers:**
  - **Color buffer**: 2D array containing colors of rasterized triangles
  - **Depth buffer aka z-buffer**: 2D array containing depths of rasterized triangles

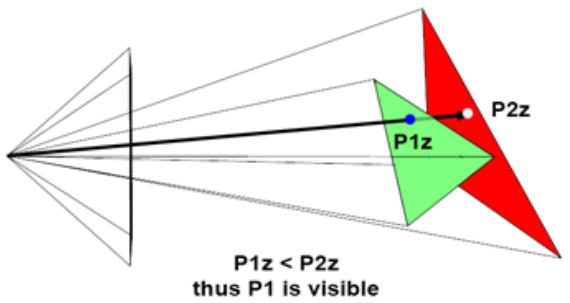




Rasterization: z-buffer

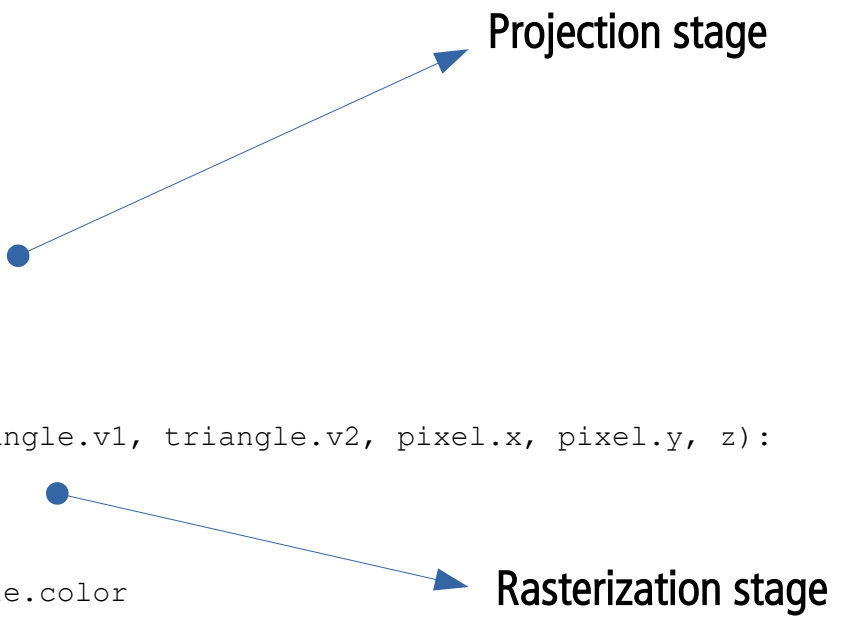
- **Multiple triangles may overlap the same pixel in the image**
- **To solve visibility, that is find closest triangle, z-buffer is used:**
  - Before rendering **z-buffer is initialized to a very large number**
  - **When pixel overlaps triangle, compute distance to this triangle** and compare it to the distance in z-buffer at that pixel position
  - **If distance to triangle is smaller then value in z-buffer**, then triangle is visible for that pixel:
    - Update z-buffer with this distance
    - Update color-buffer with color of this triangle at this pixel position
- Alternative visibility computation approach: Painter's algorithm
  - Sorting objects by distance





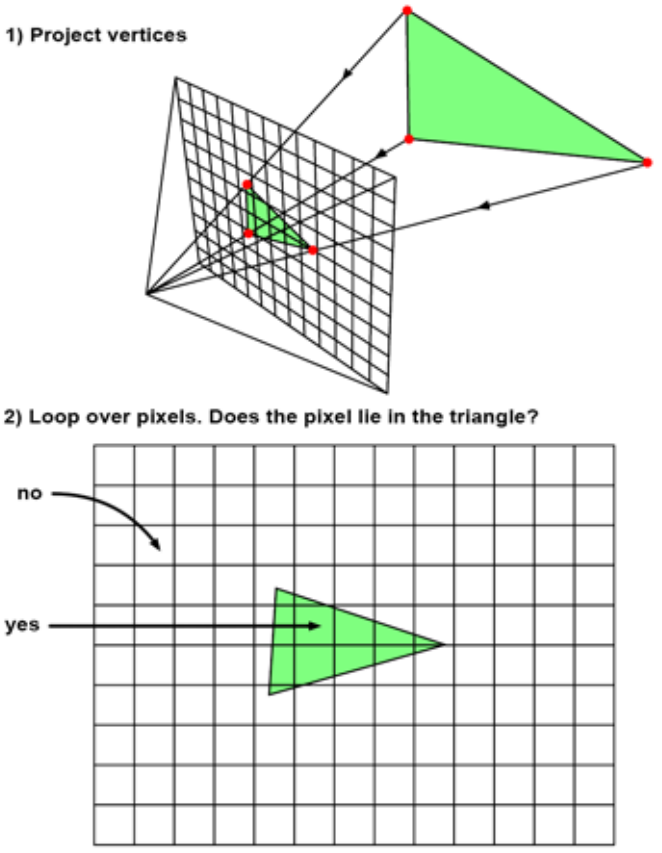
Rasterization: color-buffer and z-buffer

```
color color_buffer = new float[imageWidth, imageHeight];
float z_buffer = new float[imageWidth, imageHeight];
for (int i = 0; i < imageWidth * imageHeight; ++i):
    color_buffer[i] = color(0,0,0);
    z_buffer[i] = INF;
for each triangle in scene:
    vector v0 = perspectiveProjection(triangle.v0);
    vector v1 = perspectiveProjection(triangle.v1);
    vector v2 = perspectiveProjection(triangle.v2);
    float z;
    for each pixel in the image:
        if (pixelContainedIn2DTriangle(triangle.v0, triangle.v1, triangle.v2, pixel.x, pixel.y, z):
            if (z < z_buffer(pixel.x, pixel.y)):
                z_buffer(pixel.x, pixel.y) = z;
                color_buffer(pixel.x, pixel.y) = triangle.color
```



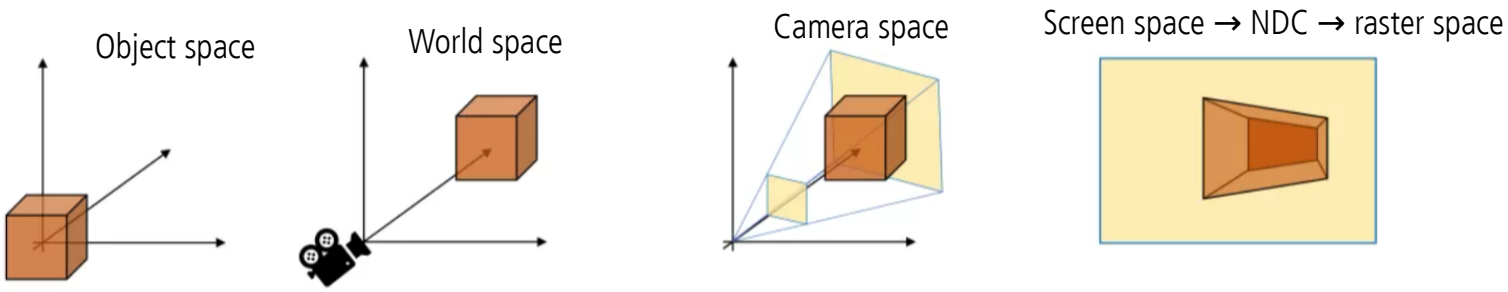
Projection and rasterization

- Projection stage
  - Project triangle vertices from world space to image plane (screen) space and convert projected coordinates to raster space
    - When triangle vertices are converted to same space as pixels it is possible to compare their coordinates
    - This way, we can find which pixels are overlapping triangle knowing only vertex positions
- Rasterization stage
  - Loop over all pixels on image plane (screen) and determine which pixels are overlapping the triangle



Projection stage

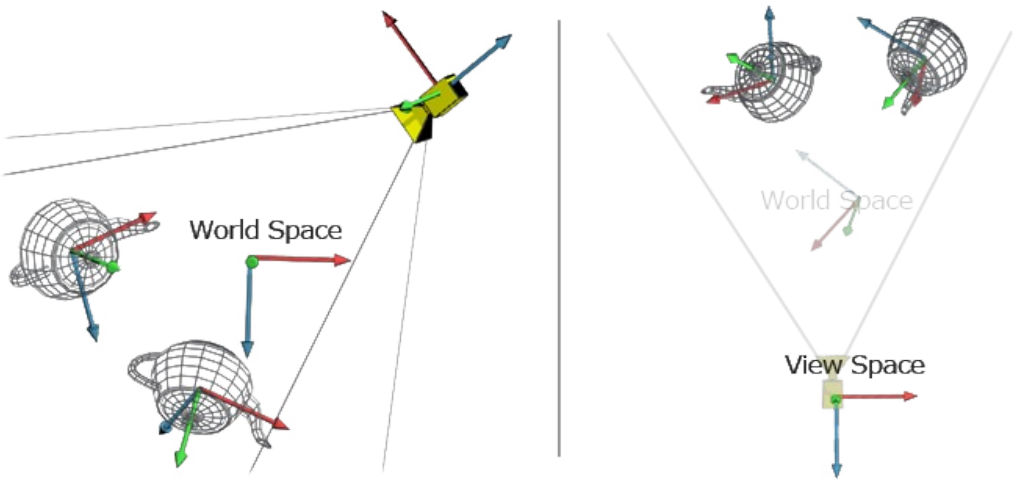
- Projection stage is responsible for **transforming triangle vertices from world space to raster space**:
  1. **World space to camera space**
  2. **Camera space to screen (image plane) space**
  3. **Screen space to normalized device coordinate (NDC) space**
  4. **NDC space to raster space**



Objects are originally defined in object space and then moved to world space.
Thus transformation from world space to camera and screen space, NDC and raster space is needed.

1. World to camera (view) space

- All triangles (object triangulated mesh) and camera are **initially positioned in world space**
- **Transform all triangle vertices $\mathtt{P}_{\mathtt{world}}\mathtt{(x,y,z)}$ and camera** so that camera is in origin, faces negative z, y points up and x points right → **view transform**
- After view transformation triangle vertices are in **camera space** $\mathtt{P}_{\mathtt{camera}}\mathtt{(x,y,z)}$



http://www.codinglabs.net/article_world_view_projection_matrix.aspx

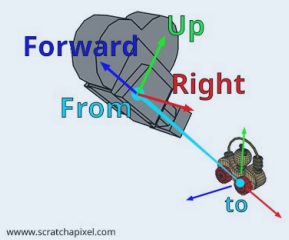
View transformation

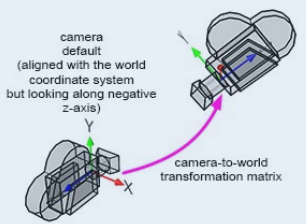
- View transformation is done using world-to-camera matrix
- World-to-camera transformation matrix can be constructed using look-at notation

$$P_{camera} = P_{world} * M_{world-to-camera}$$

inverse

$$M_{camera\text{-}to\text{-}world} = \begin{bmatrix} Right_x & Right_y & Right_z & 0 \\ Up_x & Up_y & Up_z & 0 \\ Forward_x & Forward_y & Forward_z & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$






```
Forward = normalize(From - To)
Right = crossProduct(randomVec, Forward)
Up = crossProduct(forward, right)
```

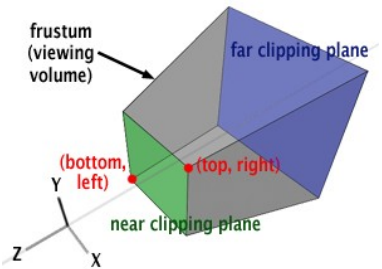

Reading: https://www.scratchapixel.com/lessons/3d-basic-rendering/computing-pixel-coordinates-of-3d-point/mathematics-computing-2d-coordinates-of-3d-points.html

2. Camera to screen space

- Project camera space vertex coordinates $P_{camera}(x,y,z)$ to screen (image plane) coordinates $P_{screen}(x,y,z)$

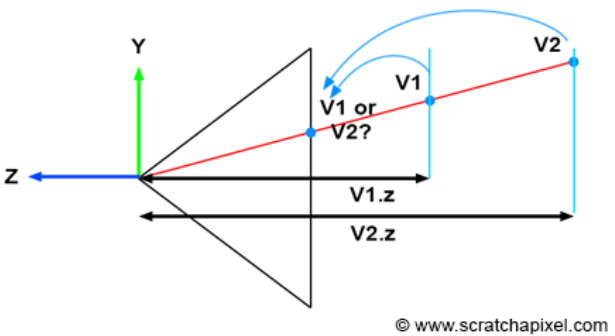
$$Pscreen.x = \frac{near * Pcamera.x}{-Pcamera.z}$$
$$Pscreen.y = \frac{near * Pcamera.y}{-Pcamera.z}$$
$$Pscreen.z = -Pcamera.z$$

- Perspective projection is used to transform points → **perform perspective divide**
- Image plane is positioned at near clipping plane at distance `near` → **multiply with `near`**

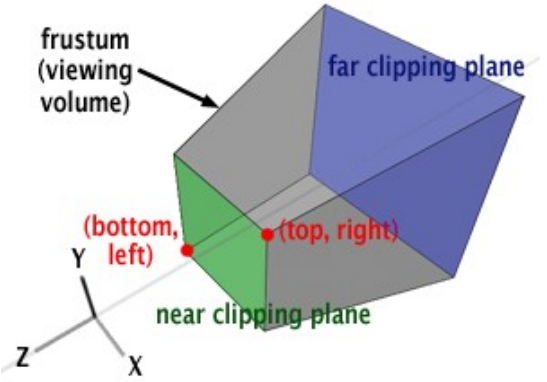


Points in screen space $P_{screen}(x,y,z)$ are still 3D:

- **(x,y) coordinates have undergone a perspective distortion**
- **z contains original vertex distance from camera** → this will be **needed for resolving visibility and overlapping triangles**



3. Screen to NDC space



`Left, right, bottom, top` are coordinates of the image plane (screen)

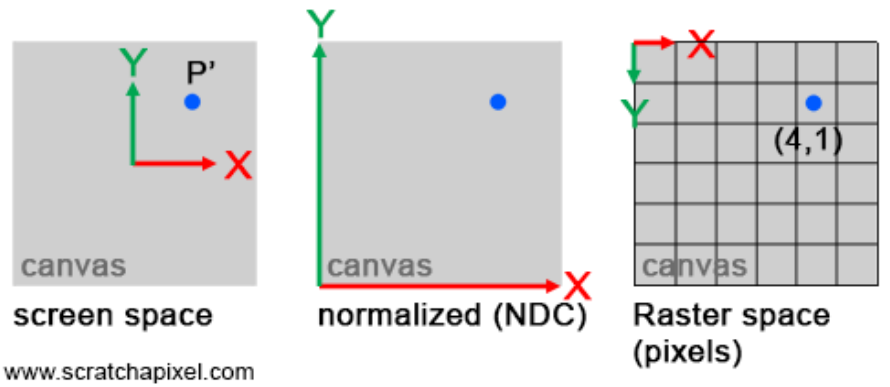
- NDC space coordinates are in range:
  - `[left,right]` → `[l,r]` for `x` coordinate
  - `[bottom, top]` → `[b,t]` for `y` coordinate
- GPU and real-time rendering standard (e.g., OpenGL, DirectX): NDC is `[-1,1]` for both coordinates
- **`(x,y)` coordinates from screen space are converted NDC space using:**

$$P_{NDC}.x = \frac{2P_{screen}.x}{r-l} - \frac{r+l}{r-l}$$

$$P_{NDC}.y = \frac{2P_{screen}.y}{t-b} - \frac{t+b}{t-b}$$

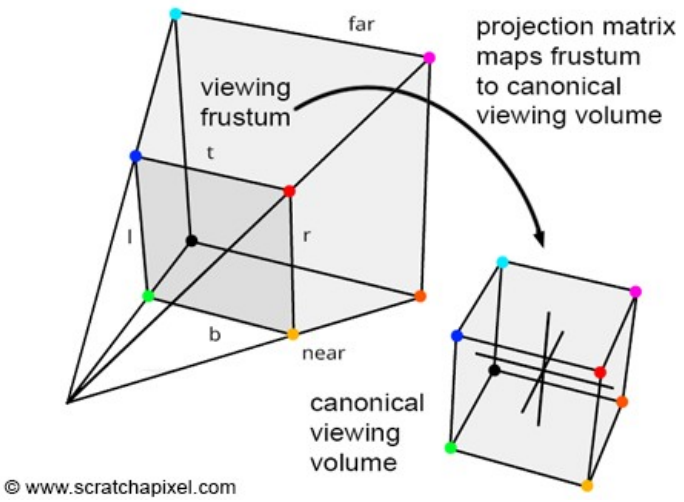
$$-1 < P_{NDC}.x, P_{NDC}.y < 1$$

- Z coordinate is remapped to [0,1] or [-1,1]
  - If P lies on near clipping plane then remap to 0 (or -1)
  - If P lies on far clipping plane then remap to 1



Note: Screen and NDC space

- Transforming vertices into screen space and further to NDC space normalizes viewing frustum into **canonical view volume a unit cube with** extreme points at `(-1, -1, -1), (1,1,1)`



Note: Screen and NDC space

- **Conversion of vertex coordinates from camera space to NDC space can also be done using projection matrix**, which contains:
  - Perspective divide operation
  - Remapping from screen to NDC space

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Perspective projection matrix is built using far (f), near (n), right (r), left(l), top (t), bottom (b) parts of view frustum which can be **calculated using angle of view and image aspect ratio.**

Read: https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/projection-matrix-introduction.html

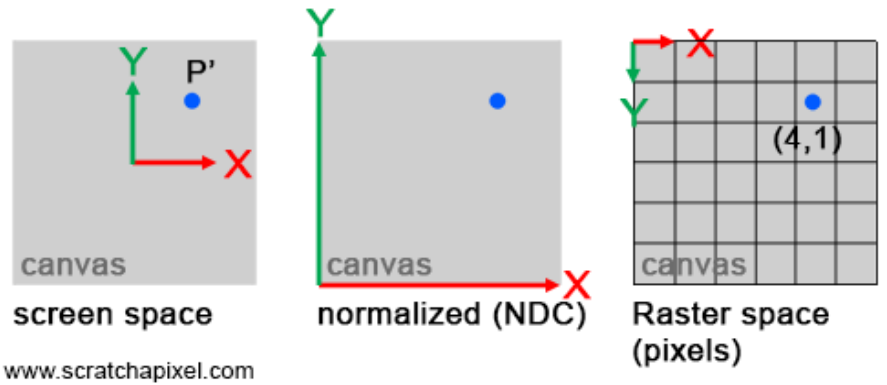
4. NDC to raster space

- **Converting coordinates from NDC to raster space (integer pixel coordinates) requires remapping** from `[-1,1] (float)` to `[imageWidth, imageHeight](int)`

$$P_{raster}.x = \frac{P_{NDC}.x + 1}{2} imageWidth$$

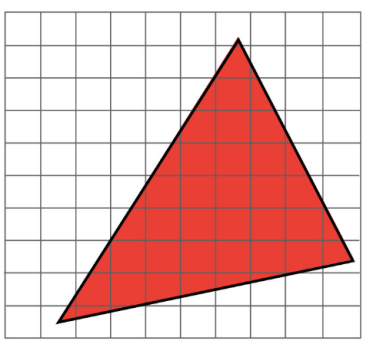
$$P_{raster}.y = 1 - \frac{P_{NDC}.y}{2} imageHeight$$

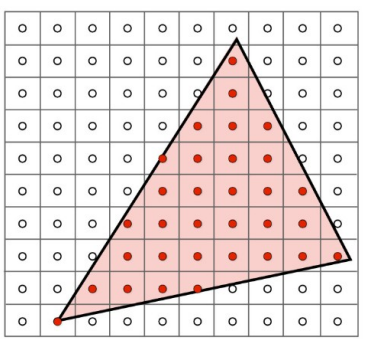
Note: In raster space the `y`-axis goes down while in NDC space it goes up. Therefore, change `y` direction during remapping process



Projection and rasterization

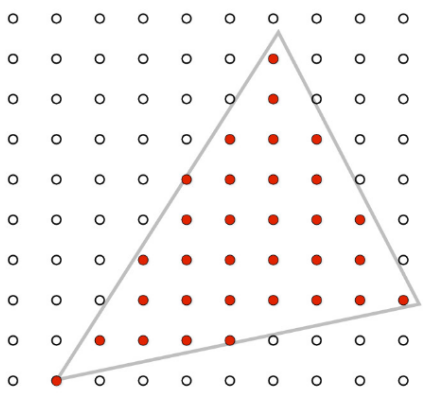
- Projection stage
  - Triangle vertex coordinates (x,y) are now converted to same space as pixels (raster space), z value contains depth
  - Now it is possible to compare pixel and vertex coordinates in order to find which pixels are overlapping triangle
    - This way, whole triangle can be rendered knowing only vertex positions
- **Rasterization stage**
  - Loop over all pixels in image plane and determine which pixels are overlapping the triangle

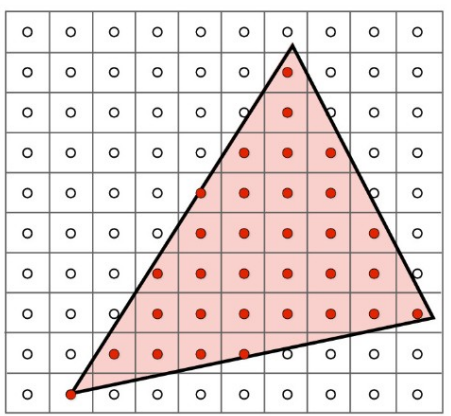




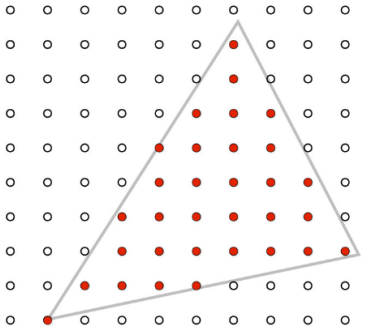
Triangle rasterization

- **Goal: convert projected triangle to a 2D image:**
  - Loop over all pixels in 2D image to determine which pixels overlap the projected triangle
    - **Inside-outside (aka coverage) test**
    - Alternative method: scanline rasterization → based on Bresenham algorithm for drawing lines
  - For each pixel that overlaps triangle, compute:
    - **Barycentric coordinates**
    - **Interpolated depth** → Needed for visibility and shading
    - Etc.





Inside-outside test: edge function

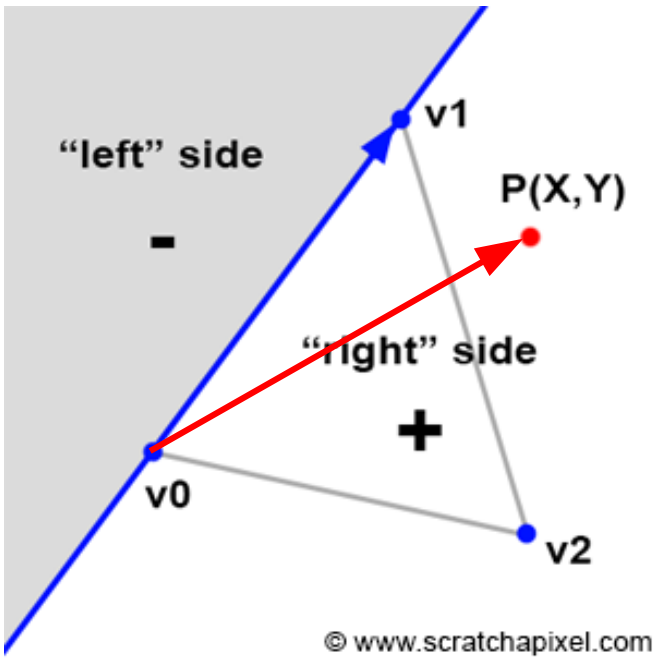


- **Triangle edges can be seen as lines splitting planes**
- **Edge function $E_{01}(P)$ returns** (clockwise winding of triangle vertices):
  - **Negative number if point P is on left side of the line**
  - **Positive number if point is on the right side of the line**

$$E_{01}(P) = (P.x - v0.x) * (V1.y - V0.y) - (P.y - V0.y) * (V1.x - V0.x).$$

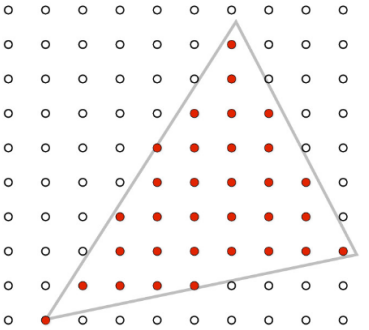
Edge function interpretation:

- Magnitude of cross product between `(v1-v0)` and `(P-v0)`
- Determinant of 2x2 matrix defined with components of vectors `(v1-v0)` and `(P-v0)`



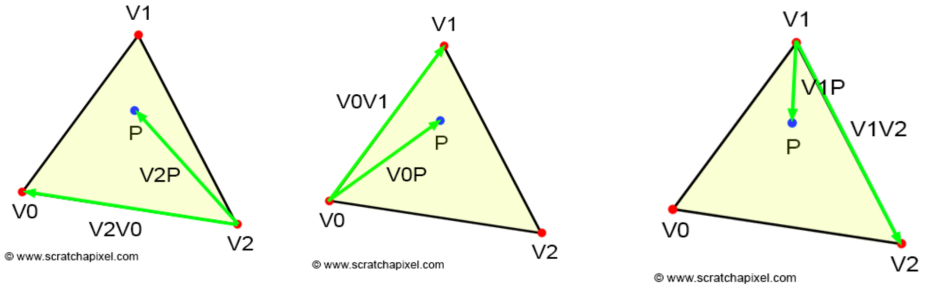
Reading: https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/rasterization-stage.html

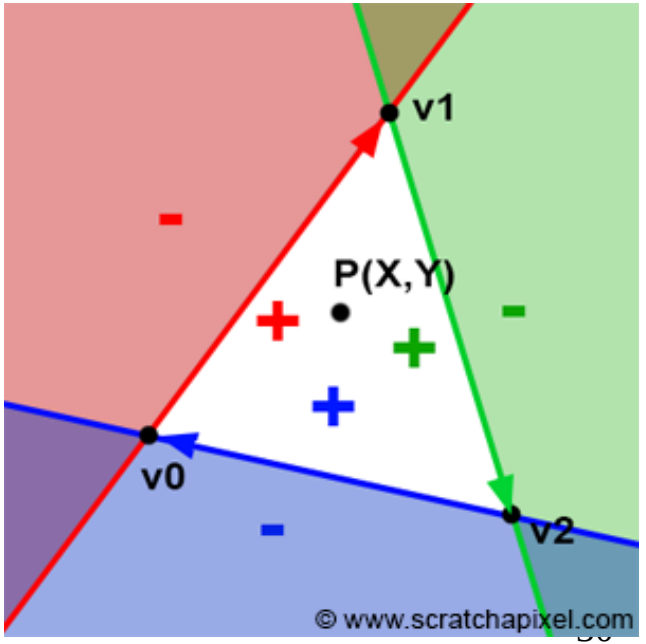
Inside-outside test: edge function test



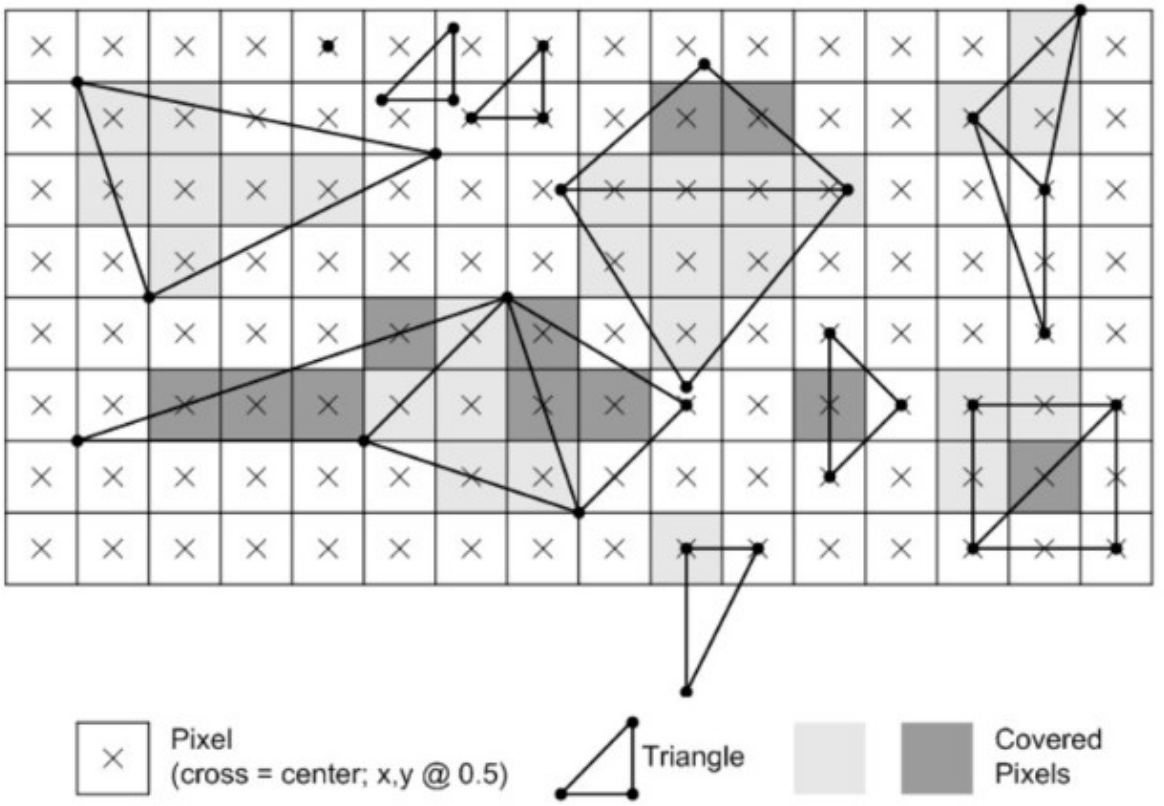
- Loop over all pixels in 2D image:
  - Pixel position P(x,y) must be tested for all 3 triangle edges
  - If edge function returns positive value for all edges, then pixel position is inside triangle
- Note: winding order of triangle matters
  - For counterclockwise winding, the sign of edge function is inverted

$$E_{01}(P) = (P.x - V0.x) * (V1.y - V0.y) - (P.y - V0.y) * (V1.x - V0.x),$$
$$E_{12}(P) = (P.x - V1.x) * (V2.y - V1.y) - (P.y - V1.y) * (V2.x - V1.x),$$
$$E_{20}(P) = (P.x - V2.x) * (V0.y - V2.y) - (P.y - V2.y) * (V0.x - V2.x).$$





Inside-outside test: example

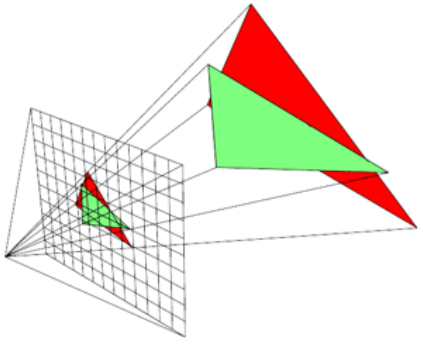


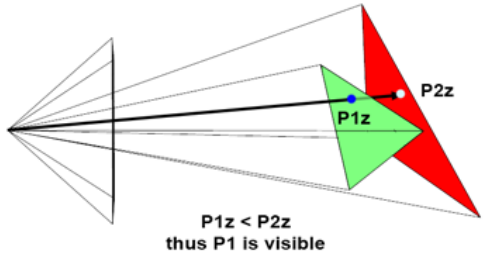
Testing if pixel is overlapping the triangle

Source: Direct3D Programming Guide, Microsoft

Depth and visibility

- Once it is determined that pixel is inside the triangle, check if it is visible
- If pixel overlaps two or more triangles use z-buffer for visibility solving
  - Compare current z-coordinate (depth) of pixel overlapping the triangle with value stored at z-buffer at that pixel position
  - If current depth value is smaller than value from z-buffer, update color-buffer with color of current triangle and z-buffer with depth of current point on triangle
- To find depth values for pixel overlapping the triangle, barycentric interpolation of depth values stored per triangle vertices is used.





Barycentric coordinates and edge function

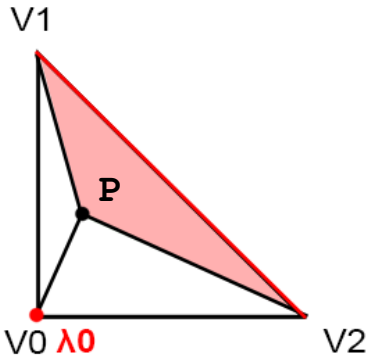
- Barycentric $(\lambda_1, \lambda_2, \lambda_3)$ coordinates can be used to define any point on the triangle.
  - Can be computed as the ratio between area of sub-triangles and the whole triangle.
  - Area of sub-triangle is calculated using edge function.

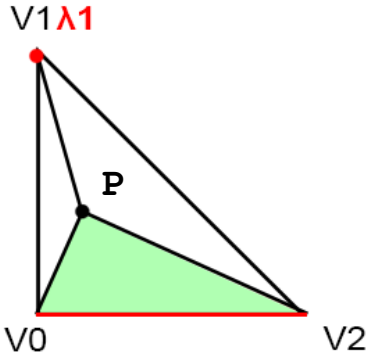
$$\lambda_0 = \frac{Area(V1, V2, P)}{Area(V0, V1, V2)},$$
$$\lambda_1 = \frac{Area(V2, V0, P)}{Area(V0, V1, V2)},$$
$$\lambda_2 = \frac{Area(V0, V1, P)}{Area(V0, V1, V2)}.$$

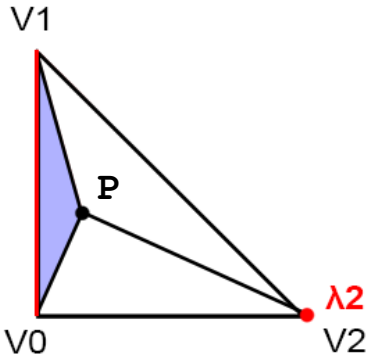
$$P = \lambda_0 * V0 + \lambda_1 * V1 + \lambda_2 * V2.$$

$$Area_{tri}(V1, V2, P) = \frac{1}{2} E_{12}(P),$$
$$Area_{tri}(V2, V0, P) = \frac{1}{2} E_{20}(P),$$
$$Area_{tri}(V0, V1, P) = \frac{1}{2} E_{01}(P).$$

$$E_{01}(P) = (P.x - V0.x) * (V1.y - V0.y) - (P.y - V0.y) * (V1.x - V0.x),$$
$$E_{12}(P) = (P.x - V1.x) * (V2.y - V1.y) - (P.y - V1.y) * (V2.x - V1.x),$$
$$E_{20}(P) = (P.x - V2.x) * (V0.y - V2.y) - (P.y - V2.y) * (V0.x - V2.x).$$



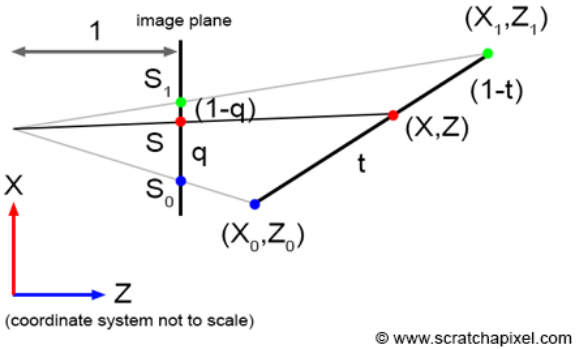




Depth and barycentric interpolation

- For current pixel overlapping triangle, z-value is found using barycentric interpolation
  - In projection stage, original camera space z-coordinate is stored for each vertex of a triangle
- **Problem: perspective distortion**: perspective projection preserves lines but not distances
- **Solution: perspective correct barycentric interpolation**: compute the inverse of P z-coordinate

$$\frac{1}{P.z} = \frac{1}{V0.z} * \lambda_0 + \frac{1}{V1.z} * \lambda_1 + \frac{1}{V2.z} * \lambda_2.$$



Perspective projection doesn't preserve distances.

Vertex attributes and barycentric interpolation

- Triangle vertices can also have assigned **vertex attributes: colors, normals, texture coordinates, etc.**
- **Vertex attributes are used during shading of pixels** overlapping the triangle
- **Barycentric coordinates** $(\lambda_1, \lambda_2, \lambda_3)$ **are used to interpolate vertex attributes** for any pixel overlapping the triangle
- **Due to distortion introduced by perspective projection**, it is required to preform **perspective correction** for vertex attribute interpolation

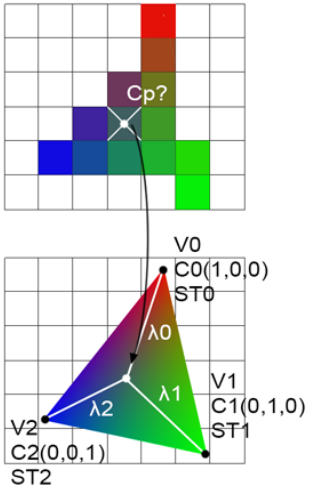
Example: interpolating color at point P where each vertex has color attribute: C_{v0}, C_{v1}, C_{v2}

~~$C_P = \lambda_0 * C_{V0} + \lambda_1 * C_{V1} + \lambda_2 * C_{V2}.$~~

$$C_P = Z(\frac{C_{v0}}{Z_{v0}}\lambda_0 + \frac{C_{v1}}{Z_{v2}}\lambda_1 \frac{C_{v2}}{Z_{v2}}\lambda_2)$$

- Z – current pixel depth
- Z_{v0}, Z_{v1}, Z_{v2} – triangle vertices depth

$C_{v0} = (1,0,0)$
$C_{v1} = (0,1,0)$
$C_{v2} = (0,0,1)$



Rasterization-based rendering on GPU

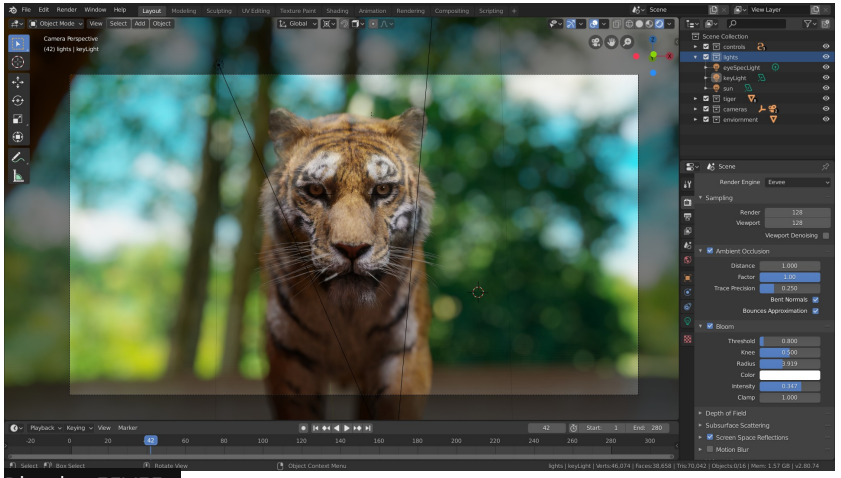
- **Rasterization is elegant algorithm for solving visibility:**
  - Projecting vertices of triangulated mesh onto image plane
  - Looping over image pixels to find which pixels lie in projected triangle
- **Both tasks are well suited for GPU and can be performed fast**
  - GPU rendering uses rasterization approach for solving visibility
  - GPU rendering is conceptually described with graphics rendering pipeline with rasterization as one module of the pipeline
  - Graphics rendering pipeline is foundation for real-time rendering

Graphics rendering pipeline: conceptual overview

Using rasterization for rendering on GPU

Motivation

- **Rasterization** is deeply integrated in **GPU hardware for rendering**.
- **Graphics rendering pipeline** conceptually describes rasterization-based rendering on graphics processing unit (GPU)
- It is used for:
  - Real-time rendering applications
  - Interactive rendering applications
  - Games, modeling, visualization, etc.
  - VR, AR, etc.



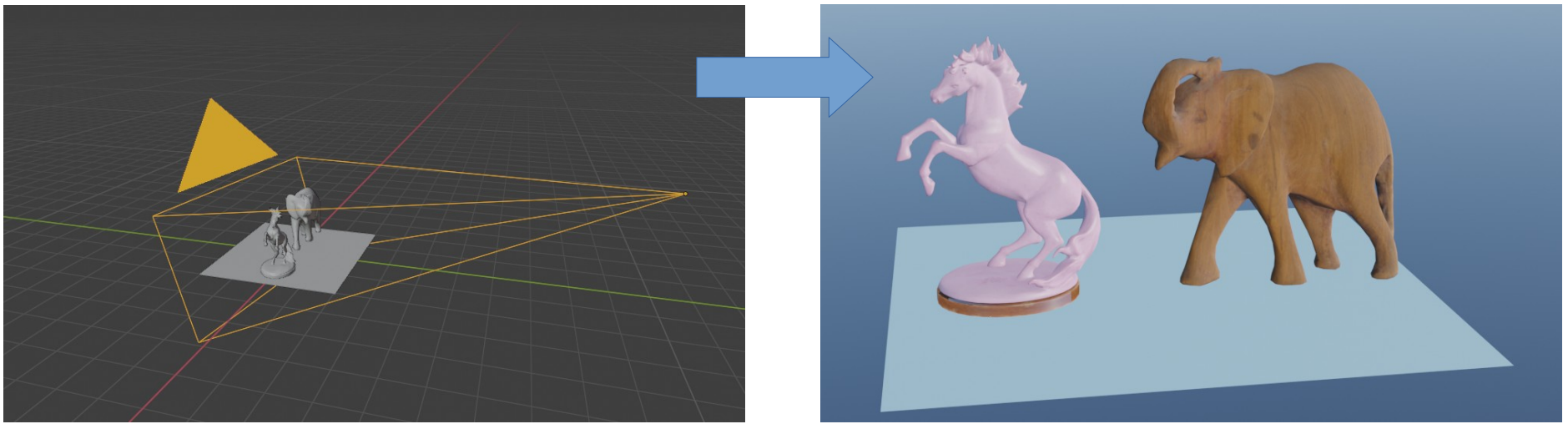
Blender, EEVEE: https://docs.blender.org/manual/en/latest/render/eevee/introduction.html

Unreal engine: https://docs.unrealengine.com/4.27/en-US/

GPU – graphics processing unit, term coined by NVIDIA to differentiate GeForce 256 from previous rasterization chips. From then on, this term is still used.

Graphics rendering pipeline

- **Main function of graphics rendering pipeline: generate a 2D image from 3D scene**
- **Rendering**:
  - **Visibility calculation:** which objects are visible from camera
  - **Shading calculation:** computing color of visible objects that is light-matter calculation and light transport

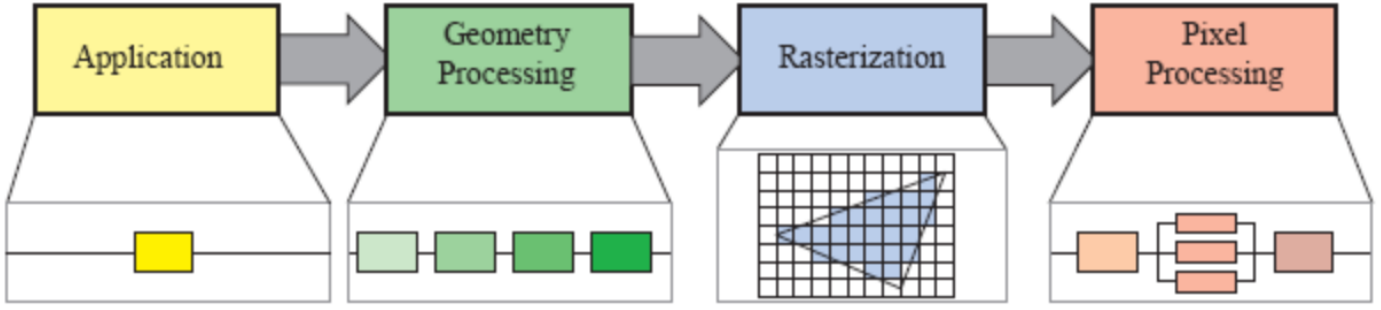


Pipeline description

- **Pipeline can be described using different abstractions levels:**
  - **Functional/conceptual stages –** describes tasks that rendering pipeline must perform but not how
  - **Physical/Implemented stages –** describes how are functional stages implemented in hardware. Physical model is up to the hardware vendor (e.g., nvidia, AMD, Intel, etc.)
  - **Logical model of GPU** –describes rendering pipeline interface exposed to a programmer by API

Graphics rendering pipeline: conceptual stages

- Graphics rendering pipeline can be **conceptually divided in four stages.**



May consists of several sub-stages

May be partly parallelized

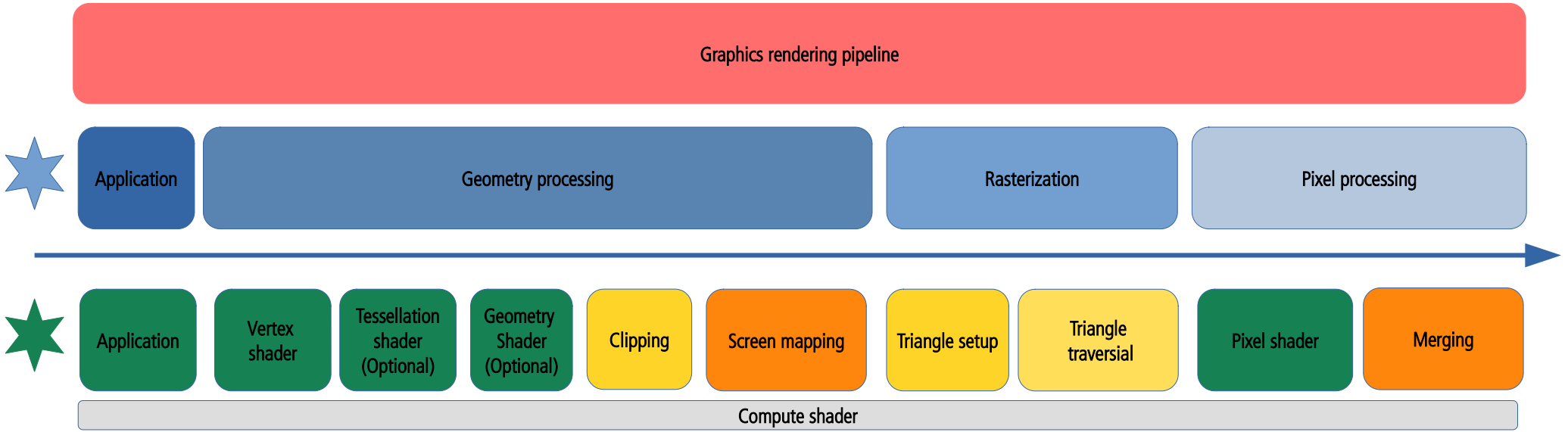
Stages can be:

- **Fixed** (no programmer control)
- **Partially configurable** (control over parameters)
- **Fully programmable (shaders)**

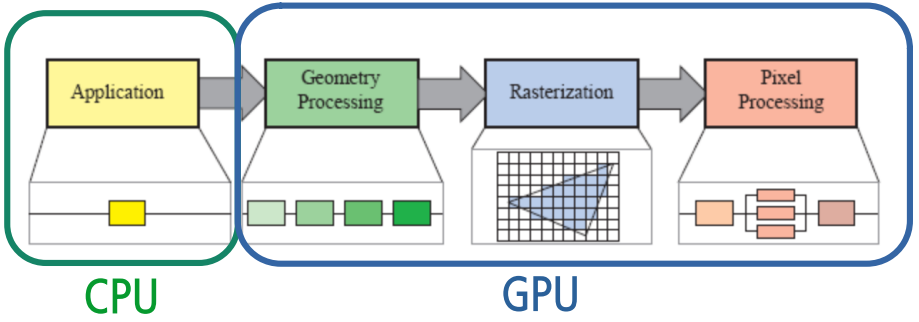
Trend is towards programmability and flexibility.

https://www.realtimerendering.com/

Graphics rendering pipeline: conceptual vs logical (API) model

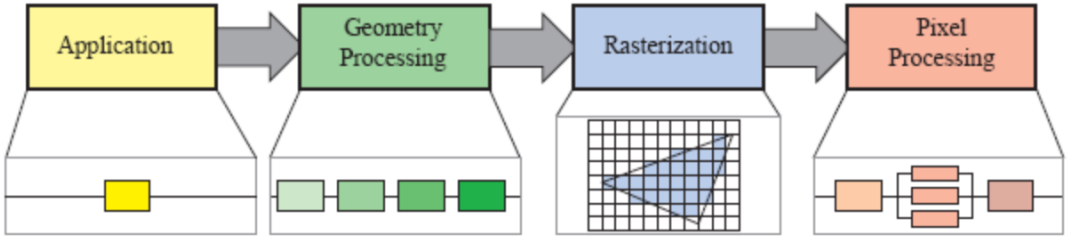


Pipeline: CPU and GPU



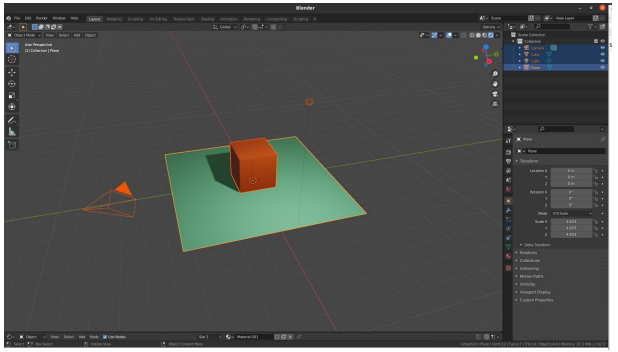
- CPU **implements application stage**.
  - CPUs are optimized for various data structures and large code bases, they can have multiple cores but in mostly serial fashion (SIMD processing is exception)
- GPU **implements** what is conceptually describes as **geometry processing, rasterization and pixel processing stage**.
  - **GPUs chips contain large set of processors called shader cores –** small processors that do independent and isolated task (no information sharing and shared writable memory) in a massively parallel fashion.
  - **Different types of programmable shaders enable controling GPU rendering**.

Pipeline: speed run



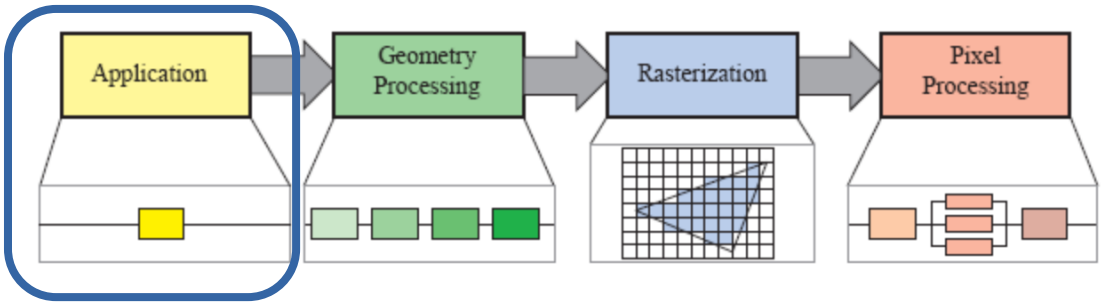
- **Application stage (CPU)**
  - Driven by application implemented in software running on CPU, e.g., 3D modeling tool application
  - 3D scene elements are defined here: objects (geometry, material), cameras, lights
- **Geometry processing (GPU)**
  - Per-triangle or per-vertex operations: transformations, projections; what, how and where it is drawn
  - Implemented on GPU that contains many programmable shader cores
- **Rasterization (GPU)**
  - Takes 3 vertices which form a triangle, finds all pixels inside triangle and forwards them to next phase
  - Fixed implementation on GPU
- **Pixel processing (GPU)**
  - Shading operation per pixel: calculating color and depth testing → programmable GPU shading cores
  - Per-pixel operations, e.g., blending new and old pixel color → partially configurable

Pipeline example: 3D modeling program



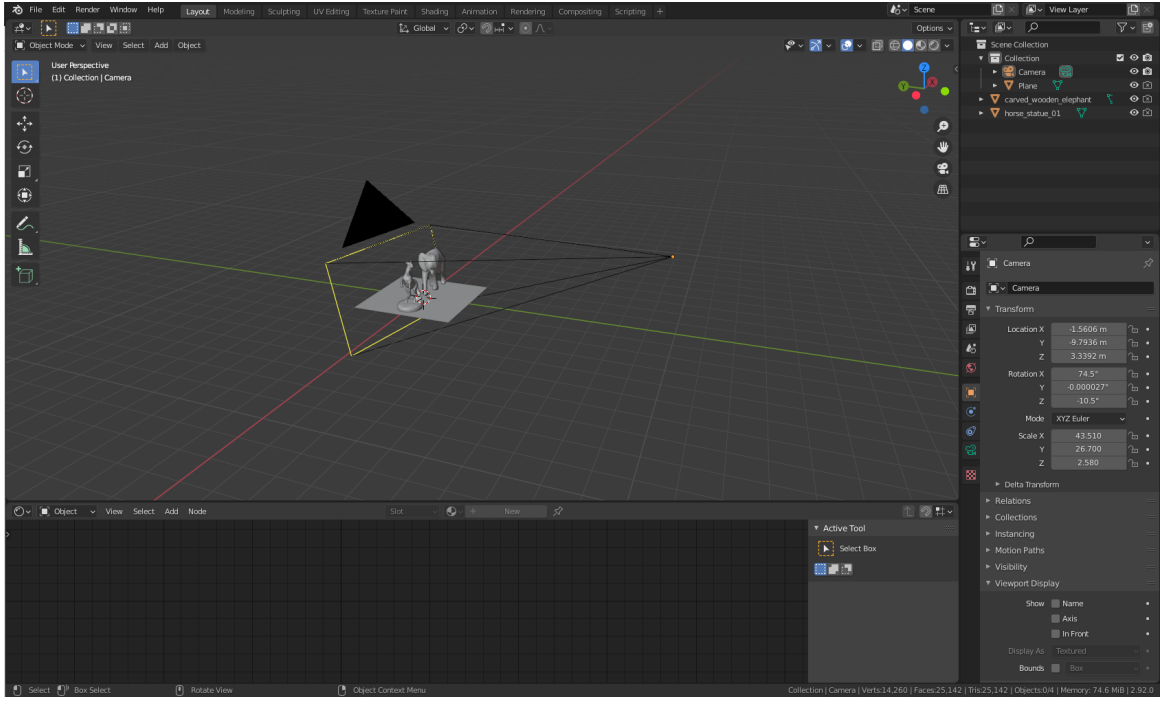
- **Application stage:**
  - Enables user to select and move parts of the model and/or camera using mouse/keyboard
    - Translate user input into transformation matrices (e.g., translation or rotation)
  - For each frame (interactive modeling) provide information to next stages for rendering: camera position, lighting and triangles of the model.
- **Geometry processing**
  - Use camera information and object transformation matrices to transform object triangles in view (camera) space
  - Project object triangles onto image plane (screen) space. Triangles not visible to camera are discarded
  - Finally, vertices are mapped into the raster space
- **Rasterization**
  - All triangles coming from geometry stage are rasterized: all pixels (fragments) inside a primitive are found and sent for pixel processing
- **Pixel processing**
  - Color of each pixel obtained from rasterization is computed using material and light information (colors, textures, shading equations) and visibility is resolved using z-buffer algorithm with optional discard and stencil testing.
  - Each object is processed and final image is displayed.

1. Application stage



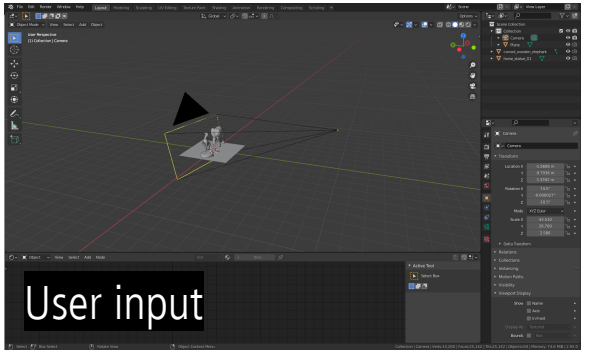
1. Application stage

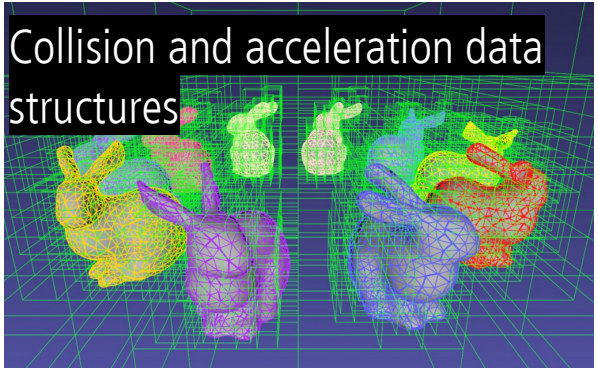
- Driven by application (e.g., 3D modeling application)
- Typically implemented on CPU (optionally on multiple threads)
- Developer has full control over what happens in this stage and how is what implemented
- Software-based implementation: not subdivided into stages



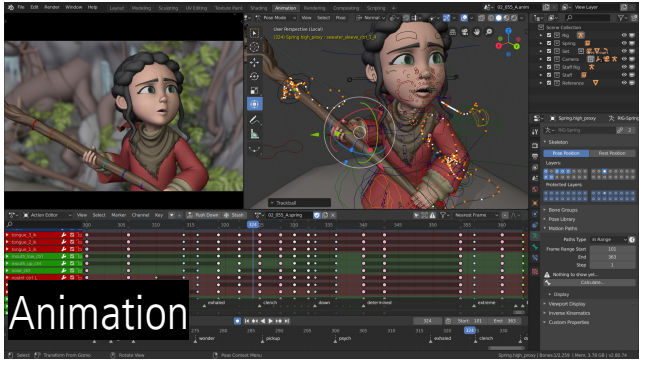
Application stage tasks

- Application stage includes tasks such as:
  - Taking care of user input from keyboard, mouse, etc. for interaction
  - Animation
  - Physics simulation
  - Collision detection – detection of collision between two objects and generating response
  - 3D scene acceleration structures (e.g., culling algorithms)
  - Compositing
  - Other tasks depending on application which subsequent stages of pipeline can not handle





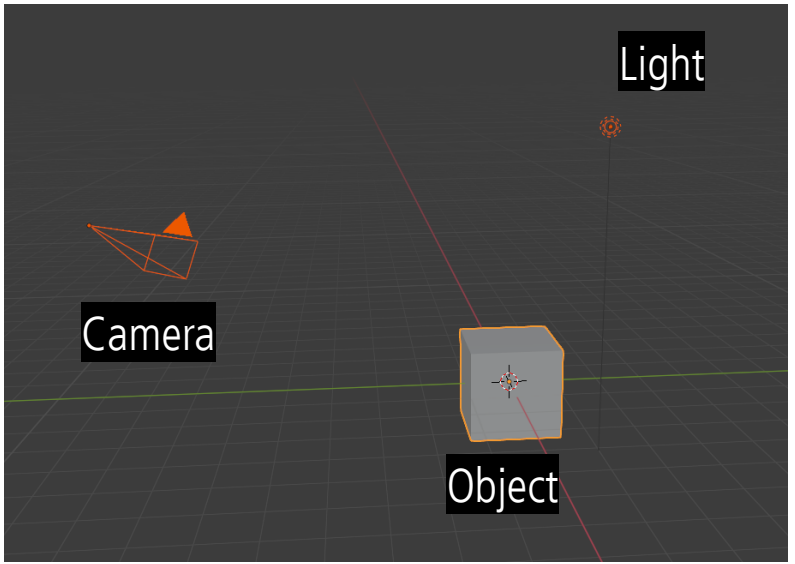
https://www.kitware.com/octree-collision-imstk/



https://www.blender.org/features/

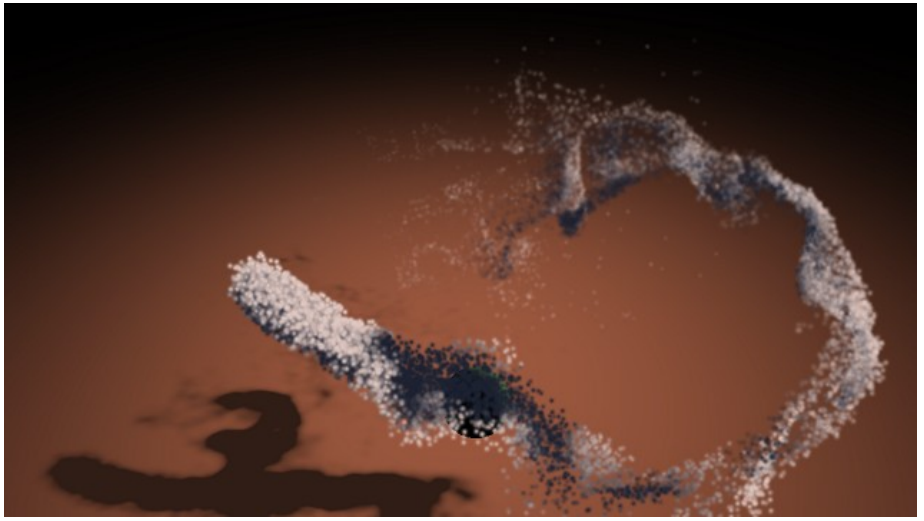
Application stage: scene definition

- On application stage elements of 3D scene to be rendered are defined:
  - **Objects**:
    - **Geometry** (shape representation): triangulated mesh, vertices, normals, texture coordinates per vertex
    - **Transformation matrices**
    - **Material** parameters (e.g., color, roughness, normal map, etc.) → per vertex
  - **Cameras**
    - **Transformation matrix** (e.g., transformation build using look at notation)
    - Camera parameters (e.g., **focal length**)
  - **Lights**
    - Position and/or direction (**transformation matrices** or vectors)
    - **Intensity, color**
    - Shape, size (geometry)

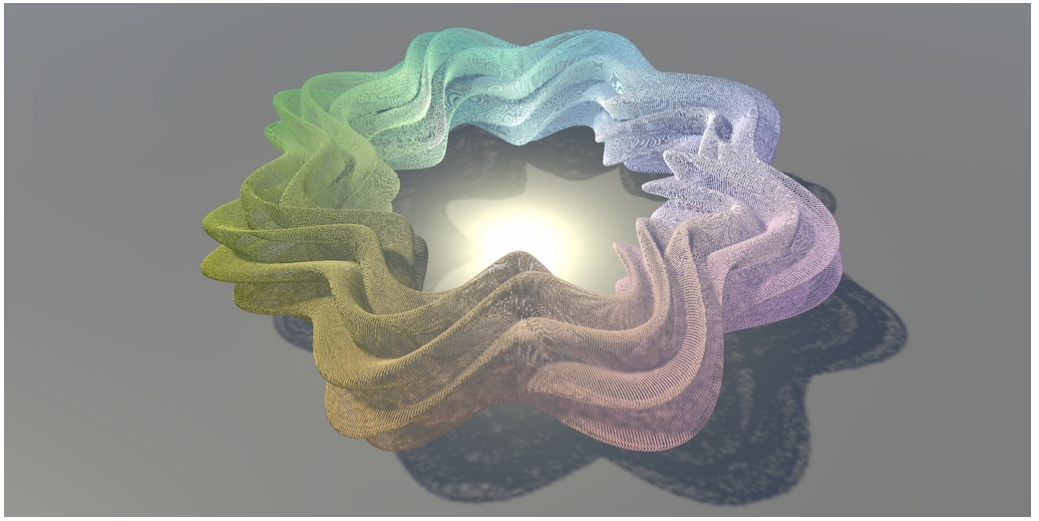


Application stage: compute shader

- Some work on application stage can be sent to GPU for processing – **compute shader**.
  - Idea treat **GPU as highly parallel general processor** rather than rendering pipeline processor



Particle simulation using compute shader:
https://arm-software.github.io/opengl-es-sdk-for-android/compute_particles.html

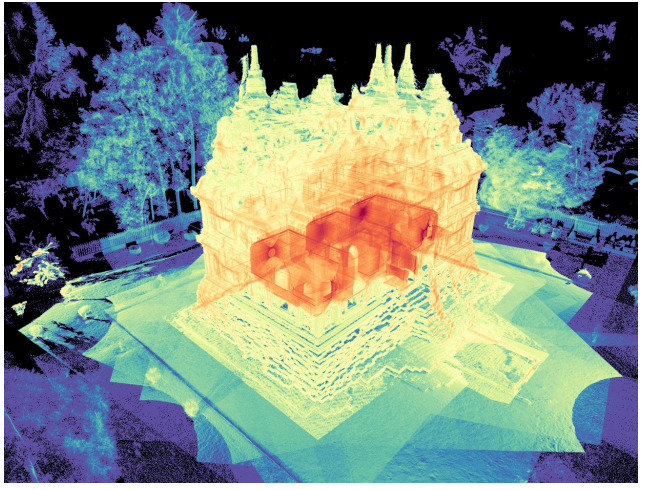


Compute shaders for point clouds animation:
https://catlikecoding.com/unity/tutorials/basics/compute-shaders/

Compute shader

https://vkguide.dev/docs/gpudriven/compute_shaders/
https://vulkan-tutorial.com/Compute_Shader
https://learnopengl.com/Guest-Articles/2022/Compute-Shaders/Introduction

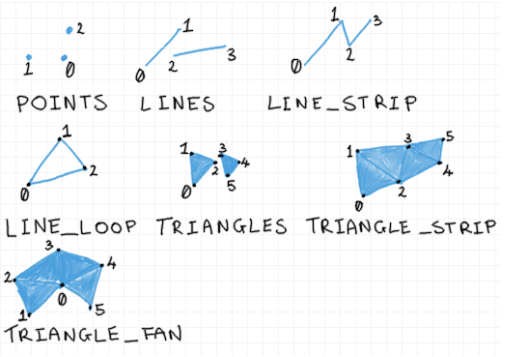
- **Compute shader is form of GPU computing** which is not necessary rendering
  - CUDA and OpenCL are used to control the GPU as massive parallel processor
- **It is closely tied to the graphics rendering but not locked into a specific location in the graphics pipeline**
  - It is used alongside vertex, pixel and other shaders
- Can be used for:
  - **Post-processing**: modifying rendered image with certain operations
  - **Particle systems**: computing behavior of particles
  - **Mesh processing**: facial animation
  - **Culling**
  - **Image filtering**
  - **Improving depth precision**
  - **Shadows calculation**
  - **Depth of field computation**
  - **Computing tessellation**



Rendering point clouds with compute shaders:
https://www.cg.tuwien.ac.at/research/publications/2021/SCHUETZ-2021-PCC/

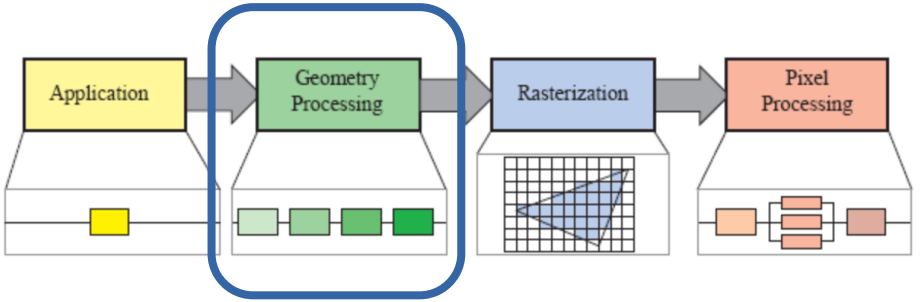
Application stage output

- **Application stage outputs data to geometry processing stage:**
  - **Object geometry to be rendered → rendering primitives:** points, lines and triangles
  - **Vectors, matrices, textures describing object materials, light and camera parameters**
- Efficiency of this stage is propagated to further into pipeline
  - Example: programmer defines amount of geometry (triangles) sent to GPU and how optimized description is



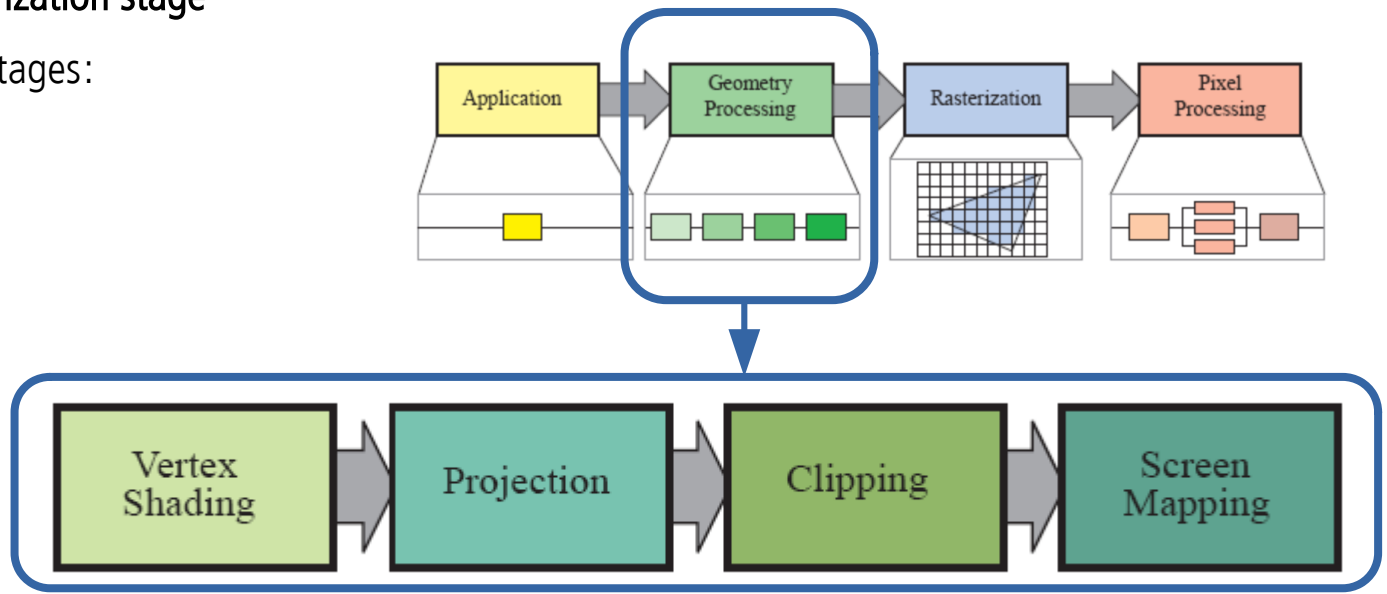
https://antongerdelan.net/opengl/vertexbuffers.html

2. Geometry processing stage

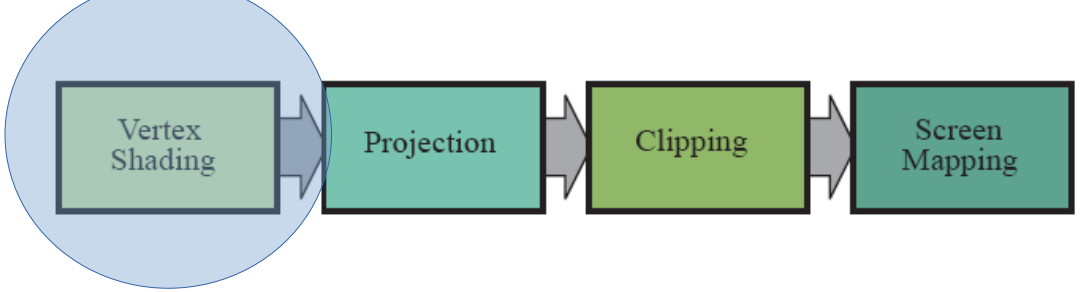


2. Geometry processing stage

- Responsible for most of the **per-triangle** and **per-vertex** geometry operations
  - Deals with **transformations**, **projections** and all other geometry handling
- **Prepares triangle vertices for rasterization stage**
- Divided into following functional stages:
  - 2.1. **Vertex shading**
  - 2.2. **Projection**
  - 2.3. **Clipping**
  - 2.4. **Screen mapping**



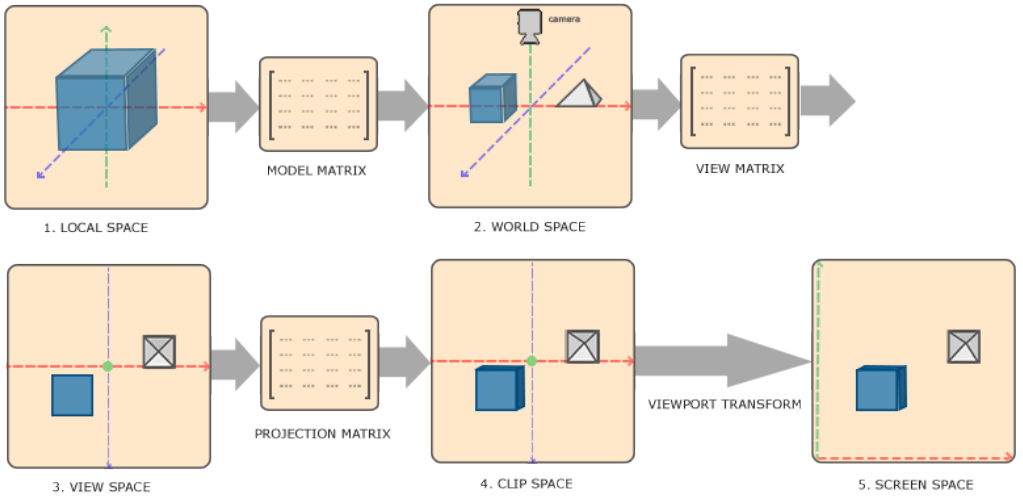
2.1 Vertex shading



- Two main tasks:
  - 2.1.1 **Transform position of vertex** inputted from application stage to space needed for projection
  - 2.1.2 **Evaluate/set-up vertex attributes**: any additional vertex data output desired by programmer such as normal, texture coordinates, colors, vertex shading, etc.

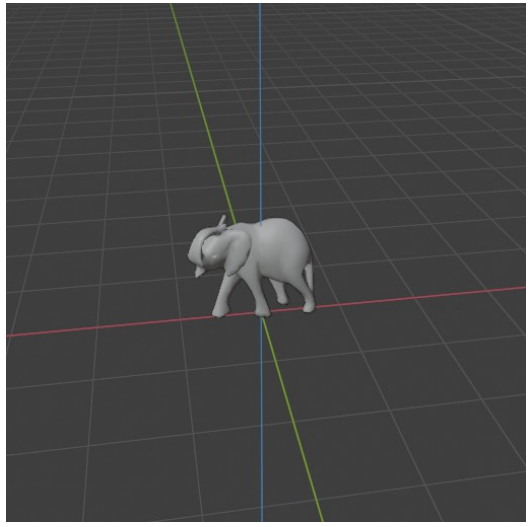
2.1.1 Vertex shading: compute vertex position

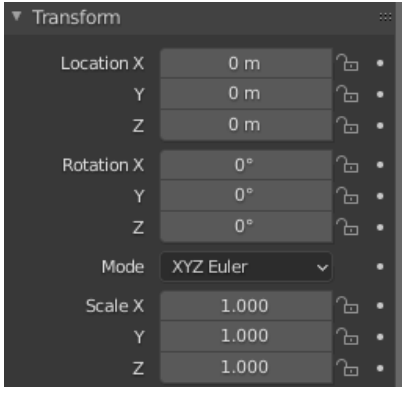
- **Vertex positions** of a 3D model (object) are minimal information that has to be passed from application stage to vertex shading stage.
- **Vertex shading performs transformation of a 3D model in several different spaces** or coordinate systems:
  - **Model/local space**
  - **World space**
  - **View/camera space**



Model space

- At start, model (vertex coordinates) resides is in its own **model space**
  - Model has not been transformed at all
- Each model can be associated with **model transform**
  - **Model transform** matrix applied on **model coordinates** gives **world space coordinates**
  - **Inverse model transform** matrix applied on **world coordinates** gives **model coordinates**
- **Model transform is 4x4 matrix applied on model's vertices and normals**
  - **Basic transformations**: Translation (T), Rotation (R), Scaling (S)
  - **Complex transformation**: Euler, Quaternions, look-at transformation matrix, etc.
  - Library: https://github.com/g-truc/glm

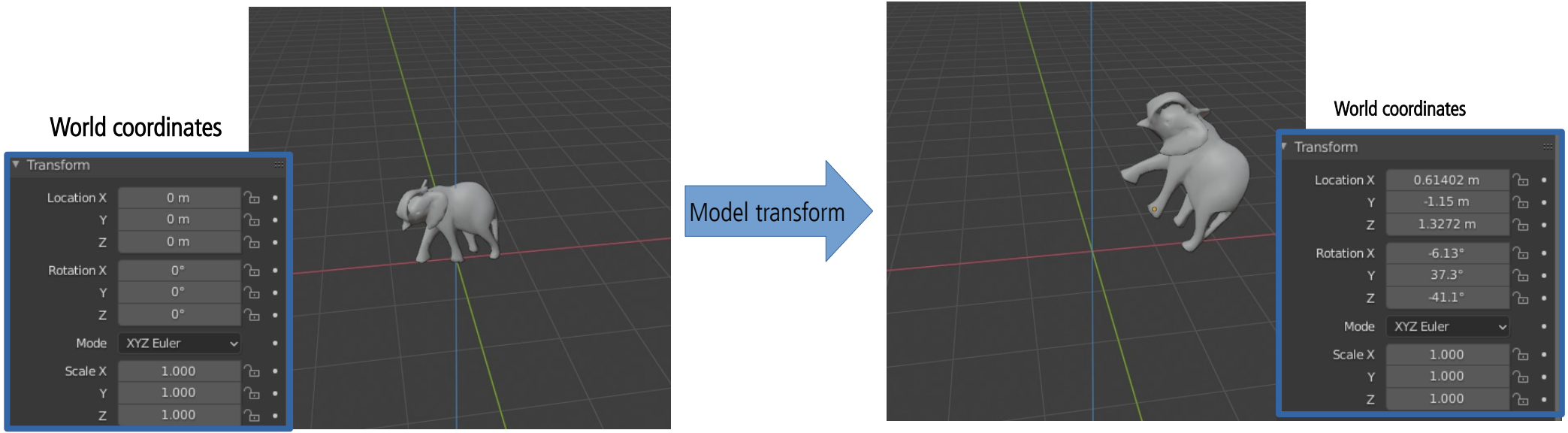




World coordinates

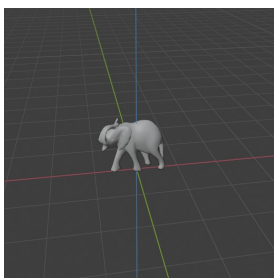
World space

- After **model transform has been applied on model coordinates**, the model is said to be in **world coordinates** or **world space**
- **World space is unique** and after all models have been transformed with their respective model transforms, they all exist in this same space
- **Model transform is defined on application stage as 4x4 matrix**, sent to GPU and applied in vertex shading stage.

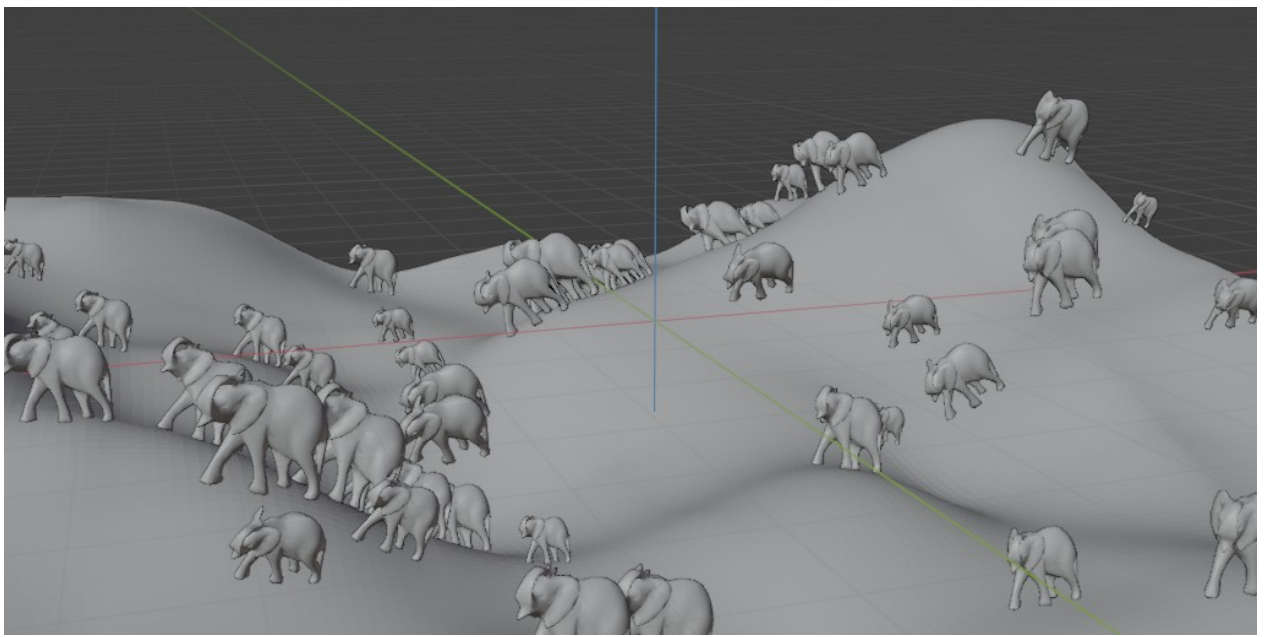


Model transform: instancing

- **Each model can have multiple transforms**
  - This allows **copying the same model across the 3D scene without specifying additional geometry – instancing**
  - Same model can have different locations, orientations and sizes in the same scene.



Same geometry is instanced using translation, rotation and scaling.



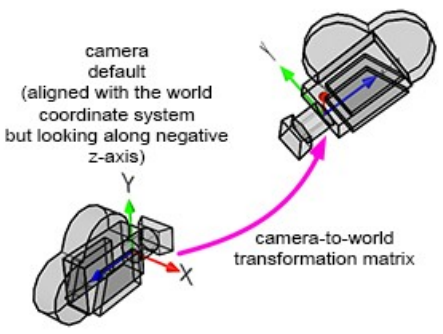
Instancing

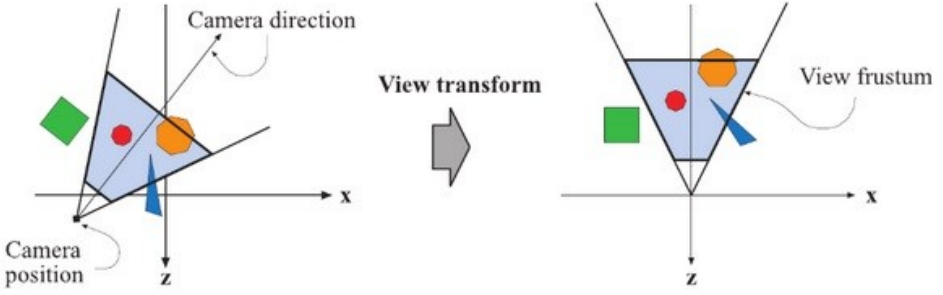
- Often used for repeated placement of same geometry with variations
  - Example: environment art

https://www.exp-points.com/junliang-zhang-procedural-environment-houdini-ue4

Camera (view, eye) space

- **After model transform, 3D model (its triangles) is in the world space, but only models visible from camera are rendered**
- **Camera has a location and orientation in world space defined with camera-to-world matrix (e.g., look-at transform)**
- Further pipeline stages require camera to be placed in world origin, aimed in negative z axis with y pointing up and x pointing right
- Therefore, **world-to-camera matrix - view transform - is applied on all model vertices and camera**
- After **view transform**, the model is said to be in **camera (view or eye) space**
- **View transform is defined as 4x4 matrix** on application stage and performed in vertex shading stage.

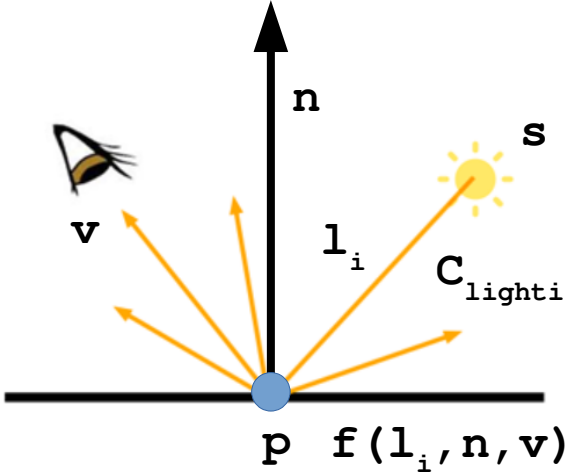




Negative z axis convention. Another convention is positive z axis convention. Actual position and direction after view transform are dependent on underlying API.

2.1.2 Computing/setting-up vertex attributes

- **Vertex attributes: colors, texture coordinates, normals, etc.**
- **Shading** can also be performed on this stage → **vertex shading**
  - Computing color is done by **evaluating shading equation** using **vertex data** (position, normal, material) and **light information** defined on application stage
  - Shading equation example: **direct illumination equation**
  - **Resulting color is stored as color per vertex** which is later **interpolated** and used across triangle and pixels
- Vertex shading was traditionally used thus the name "vertex shader"
  - Modern GPUs perform shading during pixel processing stage and vertex shader is more general unit dedicated to setting up data associated with each vertex

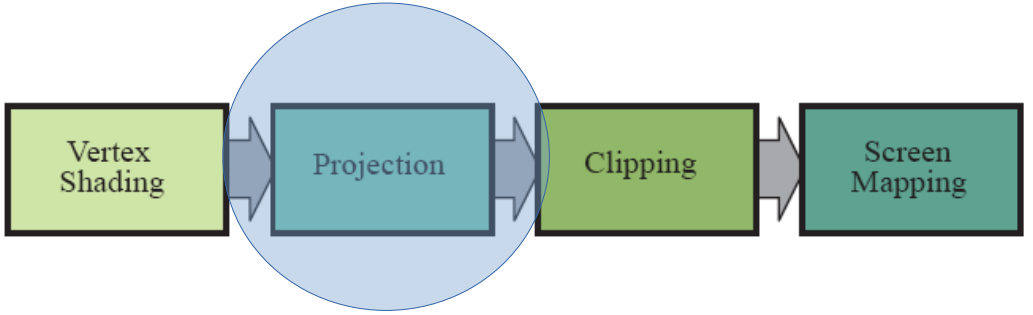


$$c_{shaded(p,v)} = \sum_{i=1}^{n} f(l_i, n, v) c_{light_i} (n \cdot l_i)^+$$

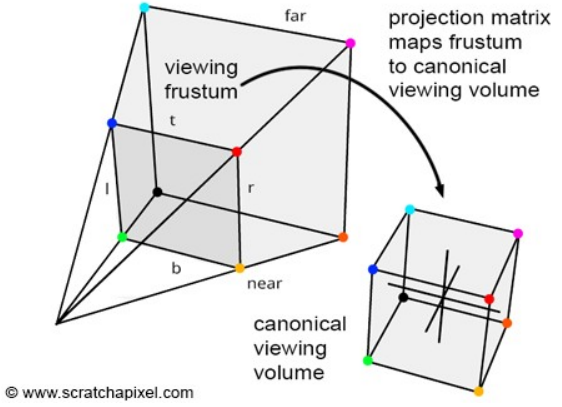
Light source color

BRDF

2.2 Projection

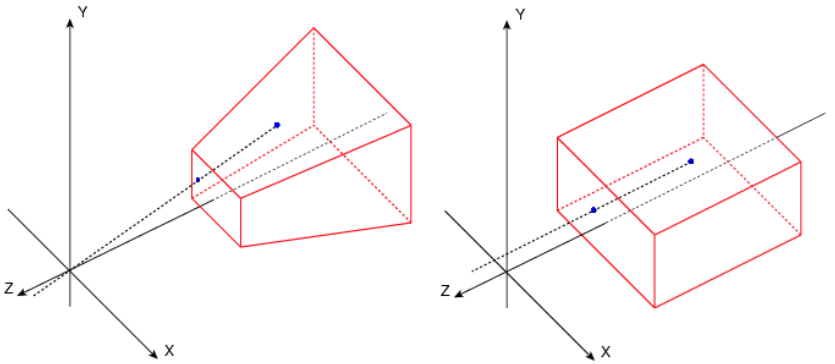


- After 3D models (their vertices) are transformed to camera space, a **projection is applied transforming vertices to clip space**
- **Projection is represented as 4x4 matrix** and can be combined with other geometry transforms: model and view
  - Projection matrix is defined in application stage and applied during geometry processing stage (on GPU) on object vertices
- **Projection transforms the view volume into canonical view volume**
  - A unit cube with extreme points at (-1,-1,-1), (1,1,1)
  - Doing so makes clipping stage easier

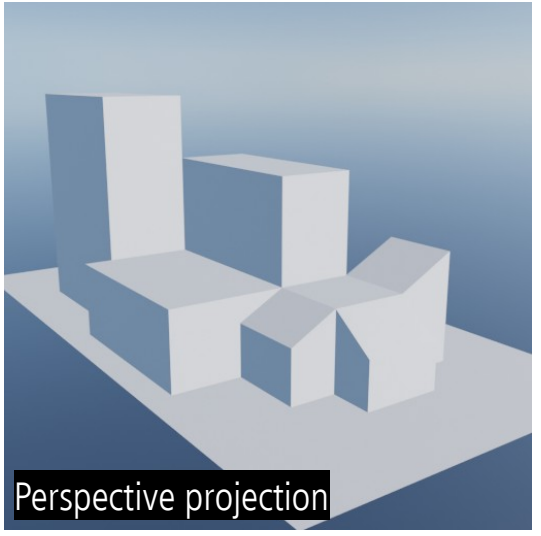


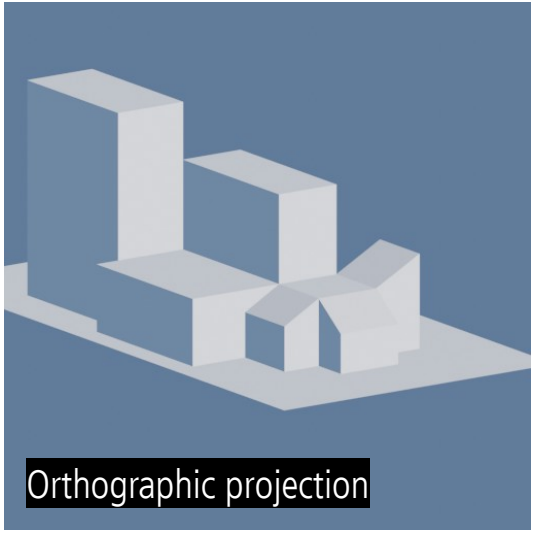
2.2 Projection

- Two commonly used projections are:
  - **Orthographic projection**
  - **Perspective projection**



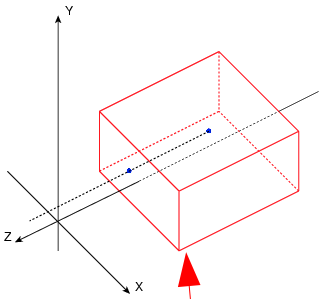
https://blender.stackexchange.com/a/649

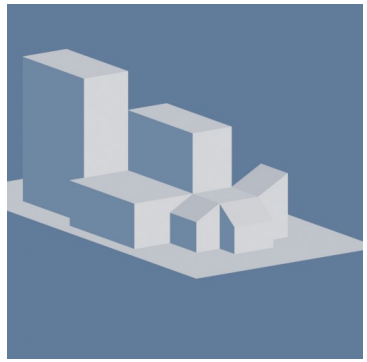




Orthographic projection

- One type of parallel projection
- **View volume is rectangular box**
- **Characteristics**:
  - **parallel lines remain parallel**
  - **Objects maintain the same size regardless of distance from camera**
- **Expressed as matrix:**
  - **Simple orthographic projection:** projects onto plane z = 0
  - **z coordinate is simply set to 0 →** store original value in depth buffer



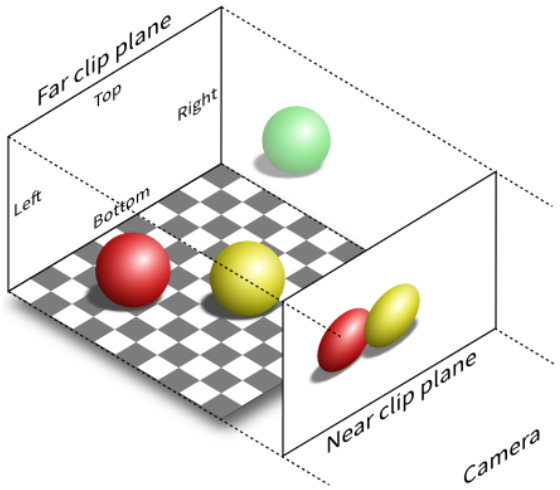


Orthographic projection

$$Pv = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ 0 \\ 1 \end{bmatrix}$$

Orthographic projection

- **General orthographic projection** is expressed using **orthographic frustum**
  - Extremes are viewing frustum **near** (n), **far** (f), **left** (l), **right** (r), **top** (t) and **bottom** (b)
  - This projection **transforms rectangular box view volume into unit cube → canonical view volume**
    - OpenGL: (-1,-1,-1), (1,1,1),
    - DirectX: (-1,-1,0) and (1,1,1)



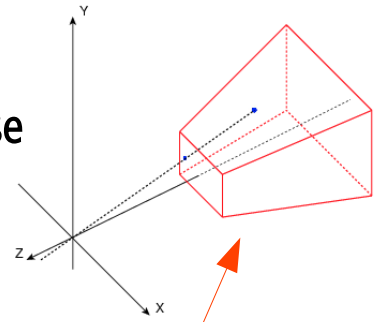
$$P = \begin{bmatrix} \frac{2}{\text{right}-\text{left}} & 0 & 0 & -\frac{\text{right}+\text{left}}{\text{right}-\text{left}} \\ 0 & \frac{2}{\text{top}-\text{bottom}} & 0 & -\frac{\text{top}+\text{bottom}}{\text{top}-\text{bottom}} \\ 0 & 0 & \frac{-2}{\text{far}-\text{near}} & -\frac{\text{far}+\text{near}}{\text{far}-\text{near}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

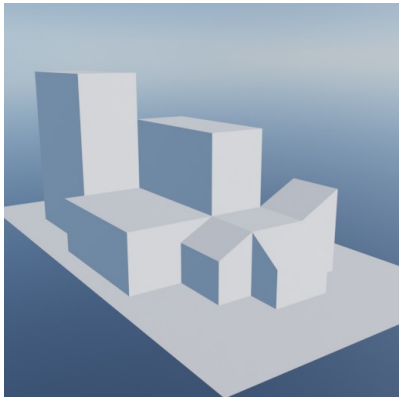
Matrix **scales** and **translates** axis aligned bounding box formed by these planes into axis-aligned cube centered around origin.

https://stackoverflow.com/q/36573283
Derivation: https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/orthographic-projection-matrix.html

Perspective projection

- View volume (frustum) is truncated pyramid with rectangular base
- Mimics how human visual system forms image
  - The further the object, the smaller appears → foreshortening effect
  - Parallel lines converge at single point
- Expressed as matrix





Orthographic projection

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Perspective projection

- **Simple perspective projection is a matrix** projecting points on a plane `z = 1`
- **After multiplication with perspective matrix:**
  - Homogeneous coordinate w_c will be equal to z vertex coordinate
  - Other vertex coordinates stay the same
- Converting to Euclidean space: **dividing by homogeneous coordinate results in Perspective divide**
  - Vertex coordinates (x,y) have undergone perspective distortion
  - Vertex coordinate z is equal to 1. Its original value is stored in depth buffer

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \frac{1}{w_c} \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix}$$

Perspective projection

- **General perspective projection matrix** is defined with viewing frustum **near** (n), **far** (f), **left** (l), **right** (r), **top** (t) and **bottom** (b)
- Camera's **field of view** (FOV) and **image aspect ratio** are used to compute left, right, bottom and top → **perspective projection matrix**

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

OpenGL perspective projection matrix: column-major order

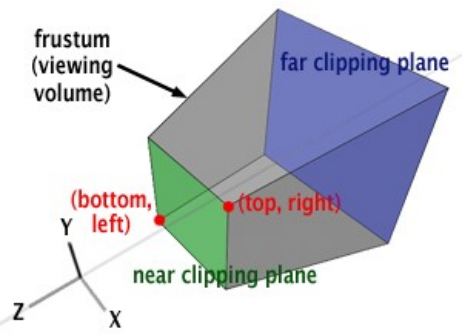
$$aspect\ ratio = \frac{width}{height}$$

$$top = \tan(\frac{FOV}{2}) * near$$

$$bottom = -top$$

$$right = top * aspect\ ratio$$

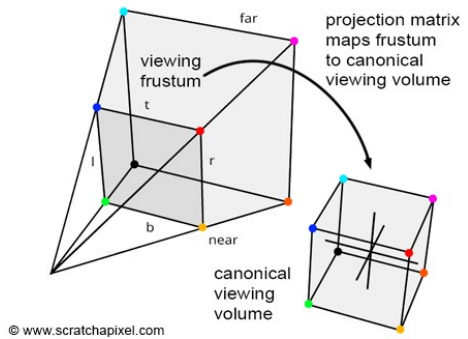
$$left = bottom = -top * aspect\ ratio$$



Derivation: https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/opengl-perspective-projection-matrix.html

Perspective projection

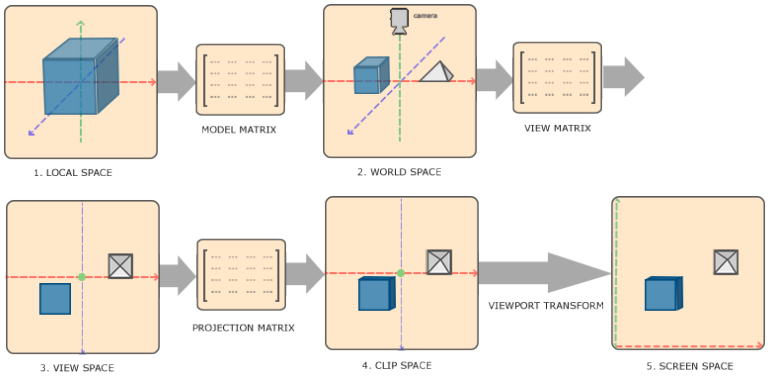
- **General perspective projection** is **matrix** transforming viewing frustum to **canonical view volume**
  - **A unit cube with extremes at** `(-1,-1,-1)` and `(1,1,1)`
- **Construction of this matrix depends on viewing furstum** and is supported by various libraries:
  - Opengl: `glFrustum(float left, float right, float bottom, float top, float near, float far);`
  - OpenGL Utility Library (GLU): `void gluPerspective(float fovy, float aspect, float zNear, float zFar);`
  - GLM: https://github.com/g-truc/glm/blob/master/manual.md#-52-glm-replacements-for-glu-functions



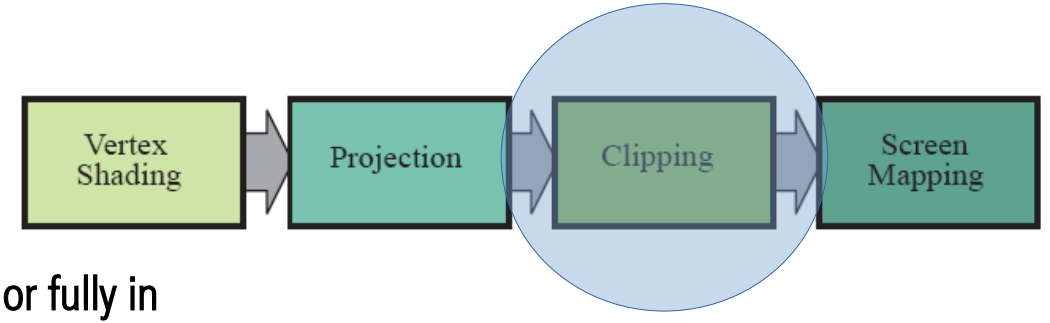
$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Clip space

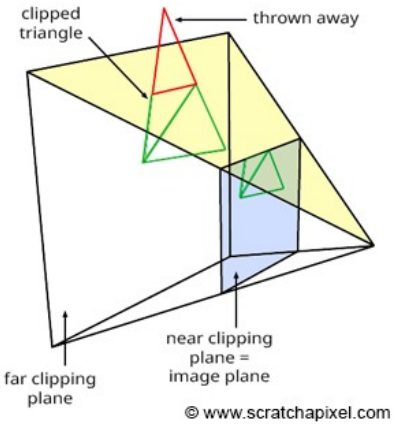
- After projection step vertex coordinates are in **clip space - clip coordinates**
  - Those are homogeneous coordinates `(x,y,z,w)`
  - Obtaining 3D (Euclidean) coordinates, requires division of `(x,y,z)` with w → **perspective division**
  - **Perspective division** to obtain Euclidean coordinates occurs **after clipping stage**

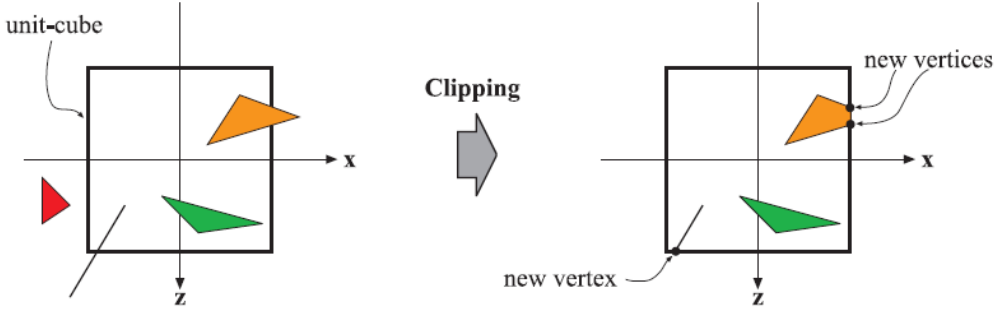


2.3 Clipping



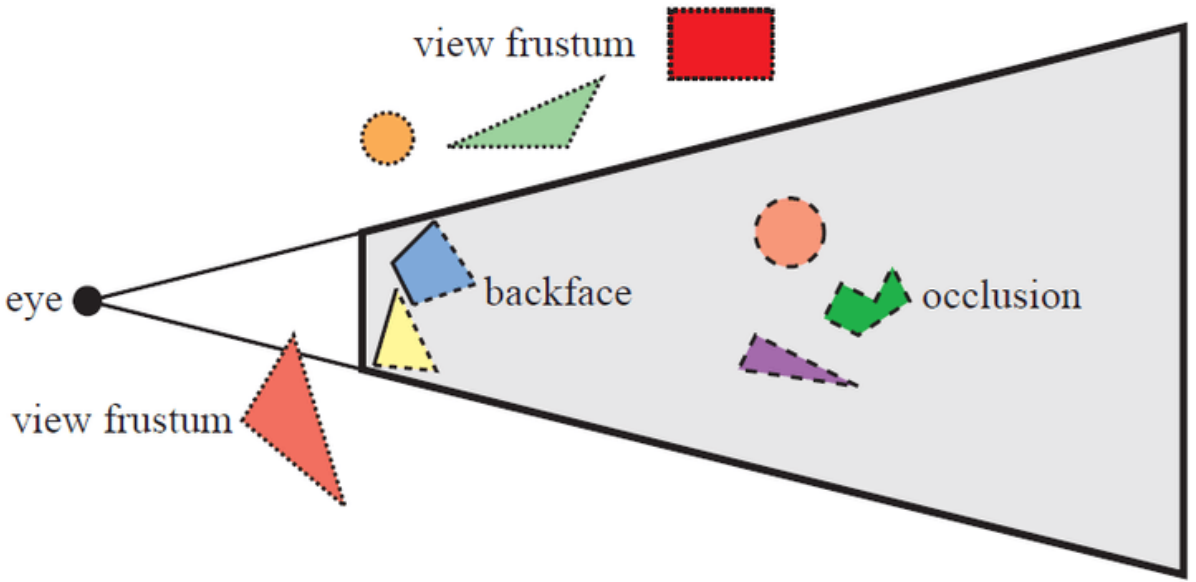
- After projection, only primitives that are **partially or fully in view volume** need to be **passed to the rasterization stage** and pixel processing for drawing on screen.
- **Primitive that is fully in view volume** will be passed further without clipping.
- **Primitives that are partially in view volume** require clipping.
  - After projection, **clipping of primitive is done against unit cube.**
  - Vertex that are outside of view volume are removed. New vertices are created on clipping position.





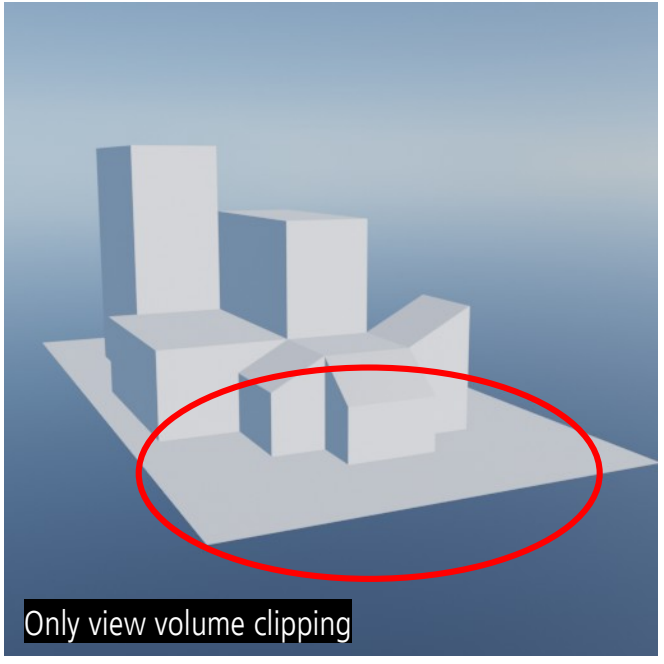
Culling

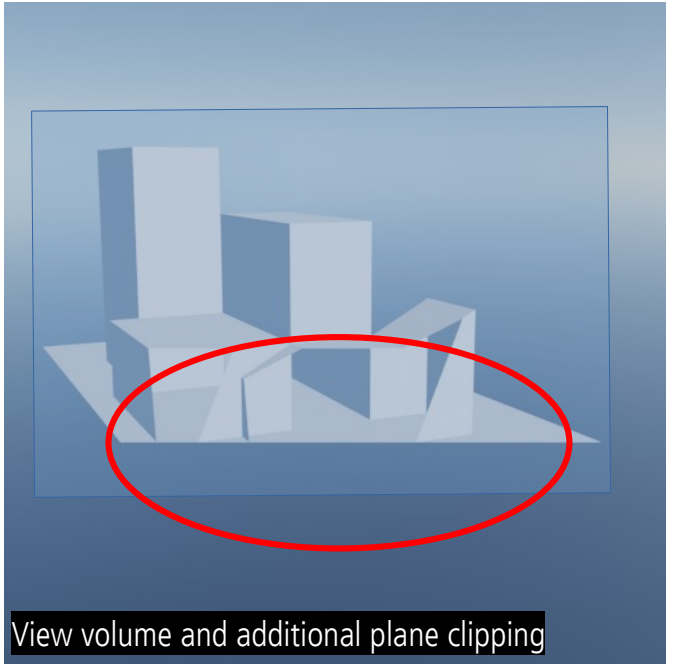
- Objects not visible by camera are discarded:
  - Frustum culling
  - Occlusion culling
  - Back face culling



2.3 Clipping

- Clipping in normally done with respect to six clipping planes of viewing frustum (view volume)
- **Additional clipping planes can be introduced by programmer** to chop the visibility of objects: **sectioning**

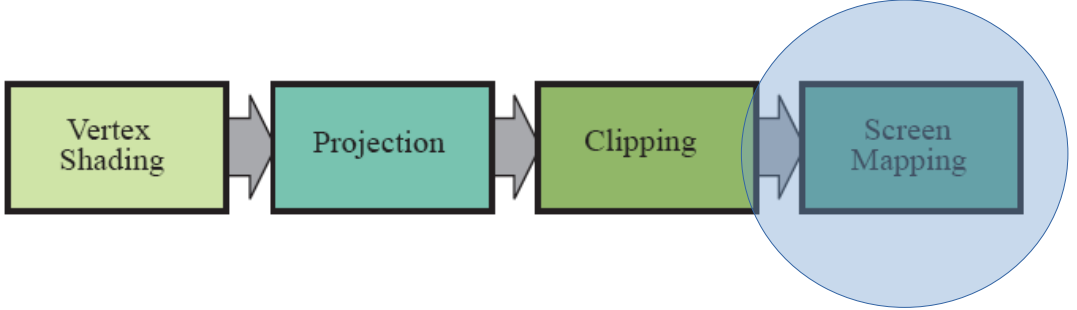




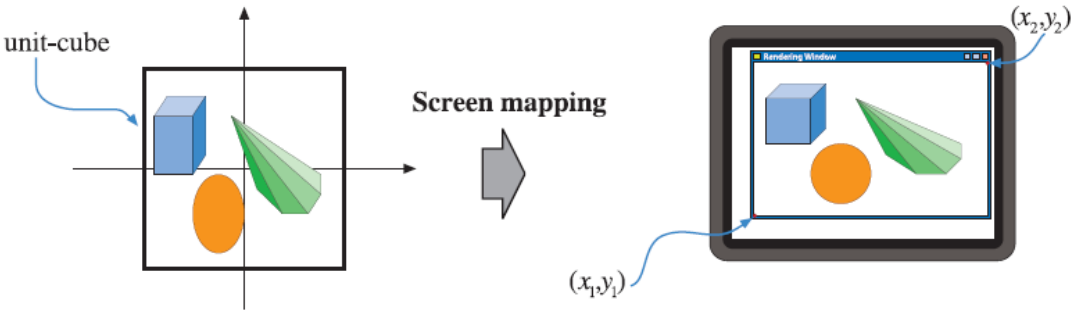
2.3 Clipping and NDC

- **Clipping uses homogeneous vertex coordinates** - clip coordinates (x,y,z,w) **resulted from projection step**
  - **Coordinate (w) contains depth (z) value**
  - **Depth value** is needed for **visibility**, **correct interpolation** of vertex attributes over triangle and **clipping**
- **After clipping, perspective division is performed:**
  - Dividing `(x,y,z)` with `w` (depth, z value)
  - Resulting triangle positions are in **canonical view volume** a 3D **normalized device coordinates (NDC)**
    - This view volume ranges from `(-1, -1, -1)` to `(1,1,1)`.
- Clipping is fixed stage in rendering pipeline → user doesn't have influence, it is done automatically
- **Resulting coordinates are sent to screen mapping stage**

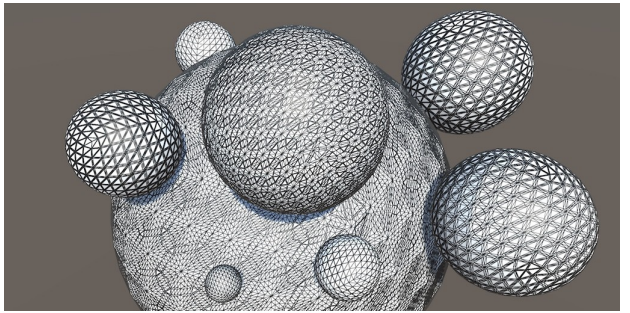
2.4 Screen mapping



- **`(x,y,z)` coordinates entering this stage are vertex NDC coordinates** `(-1, -1, -1)` to `(1,1,1)`
- `(x,y)` coordinates are mapped to raster image with dimensions (x_1, y_1) and (x_2, y_2) resulting in **raster coordinates**.
- `z` coordinate is mapped to `[-1,1]` in OpenGL or `[0,1]` in DX.
- `(x,y)` raster coordinates with re-mapped z coordinate are called **window coordinates**.
- **Window coordinates are passed to the rasterization stage.**

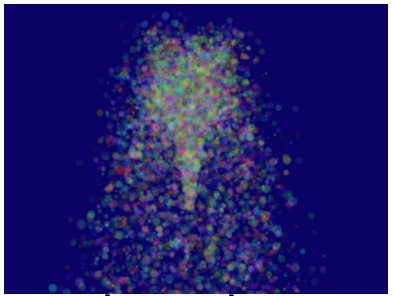


Optional vertex processing



Tessellation

- **Every GPU pipeline has the described vertex processing**
- However, there are **few optional stages that can take place on GPU**:
  - **Tessellation** – generating additional triangles to better represent curved surfaces
    - Sub-stages: hull shader, tessellator and domain shader
  - **Geometry shader** – takes in various primitives (e.g., points, triangles) and creates new vertices
    - Often used for particle generation – e.g., for each input vertex representing particle position generate additional vertices to create square on which texture can be mapped
  - **Stream output** – instead of sending vertices down the pipeline, use them for further processing on CPU/GPU
    - Often used for particle simulation
- Note: not all GPU hardware support those stages

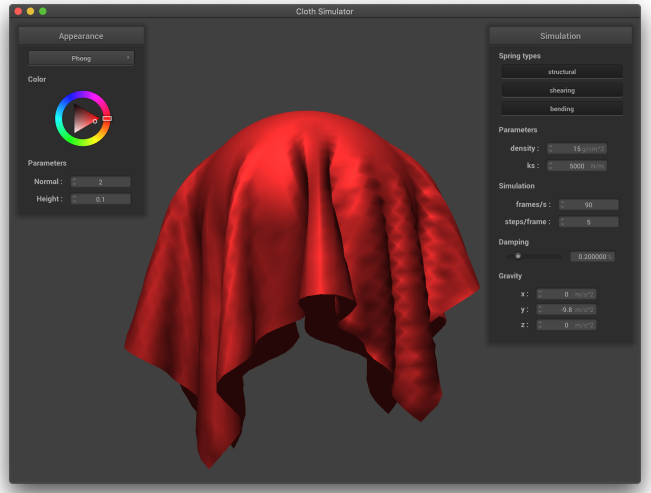


Particle simulation

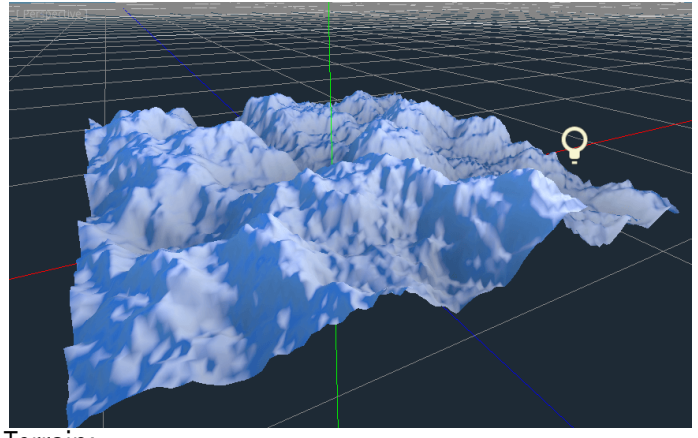
https://catlikecoding.com/unity/tutorials/advanced-rendering/tessellation/
http://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/particles-instancing/

Vertex processing applications

- **Animating character's body and face**: skinning, morphing, etc.
- **Procedural animation and deformations**: cloth, water, etc.
- **Instancing objects**: copying same object across 3D scene using different transformation matrices
- **Particle creation**: creating triangle mesh (particle) from input vertices
- **Effects**: lens distort, heat haze, water ripples, page curls, etc.
- **Terrain modeling**: applying height fields given by texture
- Etc.

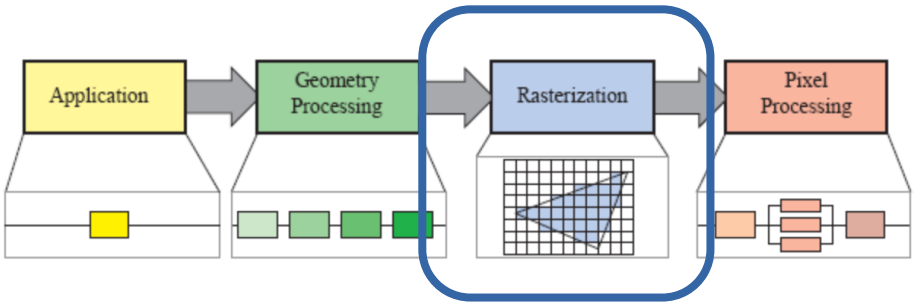


Cloth simulation:
https://andrewdcampbell.github.io/clothsim/



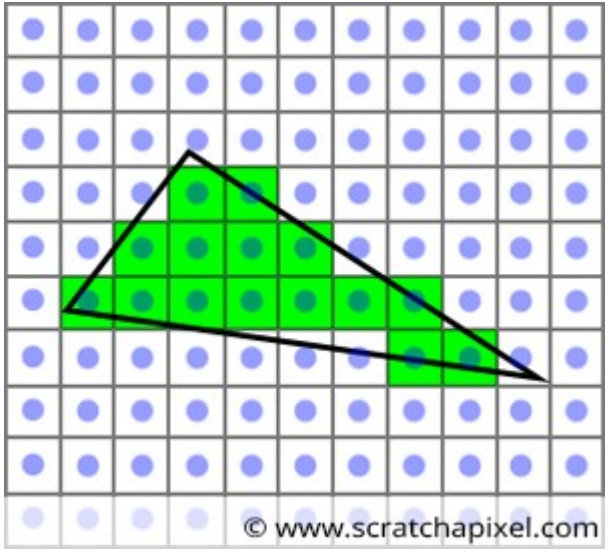
Terrain:
https://docs.godotengine.org/en/3.0/tutorials/3d/vertex_displacement_with_shaders.html

3. Rasterization stage



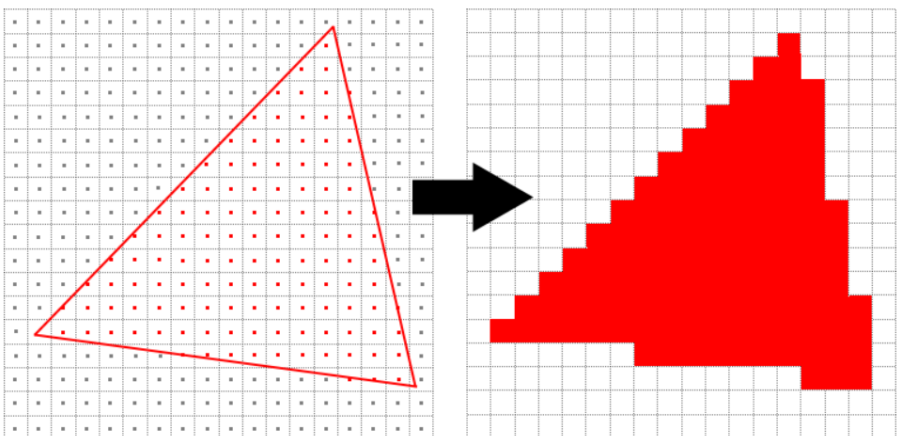
3. Rasterization stage

- Given transformed and projected vertices (with associated vertex attributes data for shading) the goal is to **find all pixels which are inside the primitive** (e.g., triangle) to be used in pixel processing stage.
  - Typically takes input of 3 vertices forming a triangle, finds all pixels that are considered inside the triangle and forwards those further for pixel processing
- Rasterization (aka screen conversion) is **conversion from 2D vertices in screen space into pixels on the screen**.
  - These 2D vertices have associated z value (depth) and vertex attributes for shading (e.g., normals)

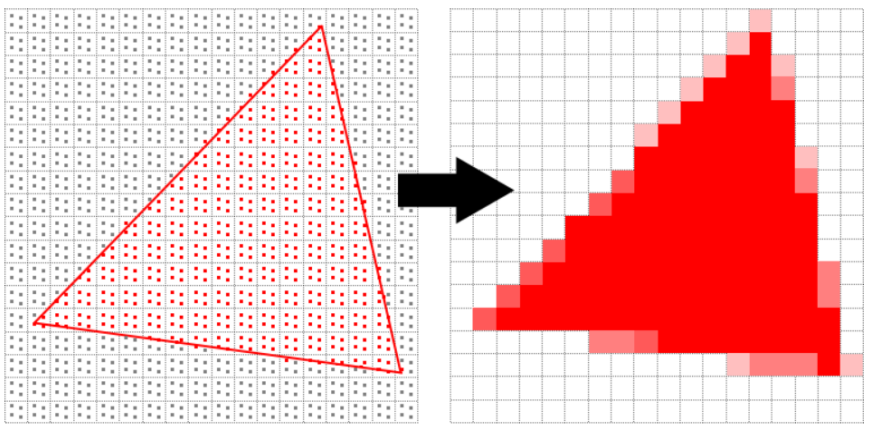


3. Rasterization stage

- **To test if triangle is overlapping a pixel** then a **pixel point sampling** is needed:
  - **Only center of pixel**: if center of pixel is inside triangle → pixel is considered inside triangle
  - **Multiple samples per pixel** are desired to evade **aliasing** problems → **super-sampling** or **multi-sampling anti-aliasing.**
    - **Each tested pixel sample inside triangle is called fragment and it contributes to color of the pixel**
- Another approach is to define pixel overlaps triangle if at least part of pixel overlaps triangle.



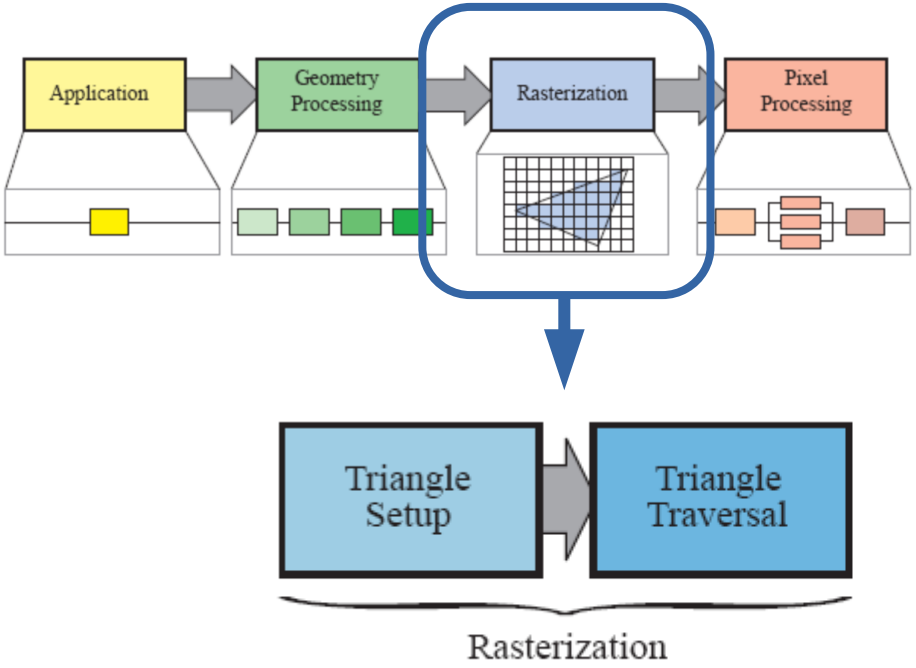
https://vulkan-tutorial.com/Multisampling



MSAAx4

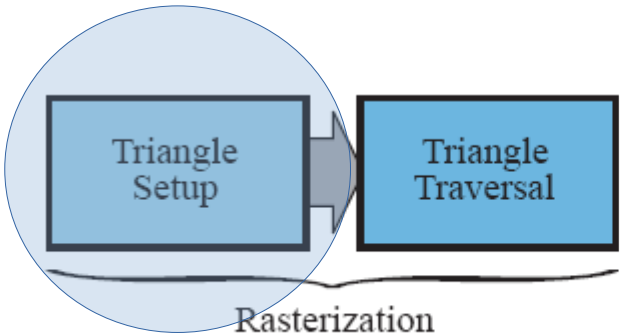
3. Rasterization stage

- **Two functional sub-stages:**
  - **3.1. Triangle setup**
  - **3.2. Triangle traversal**
- Process of "synchronization" between geometry processing and pixel processing
- This stage can also process lines and points but triangles are most often used

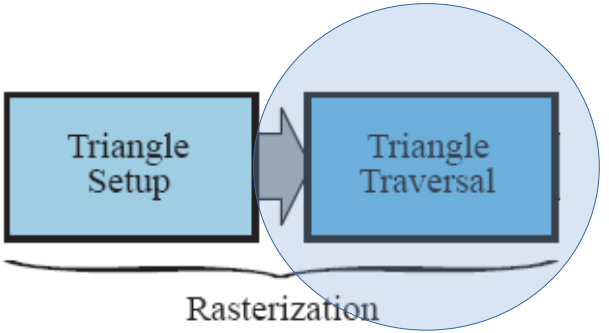


3.1 Rasterization: triangle setup

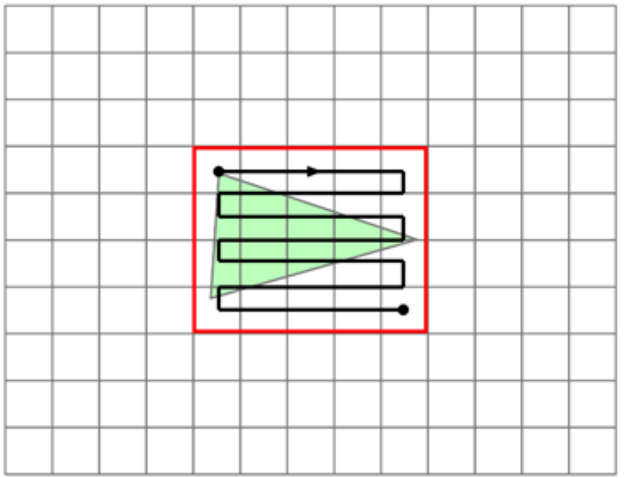
- In this step all **data for the triangle needed for sub-sequent stages are computed using triangle information**
  - **Edge function** needed for inside outside test
  - **Barycentric coordinates** and data needed for interpolating depth values and vertex attributes
- Computed data is needed for:
  - Triangle traversal and inside outside test
  - Interpolation of shading data produced by geometry stage (e.g., normals)
- Fixed-function hardware is used for this stage → programmer has no control over it.



3.2 Rasterization: triangle traversal

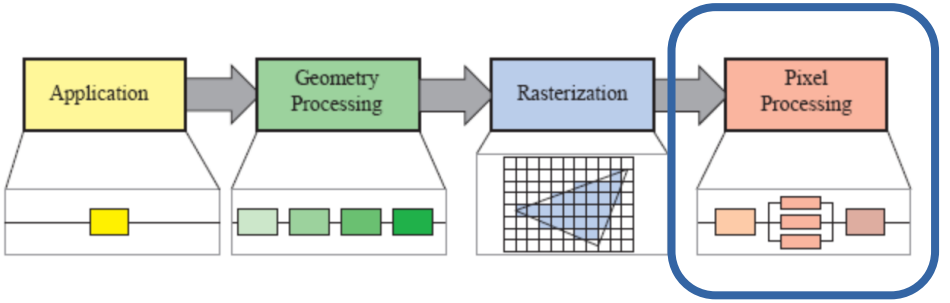


- **Triangle traversal**: each image pixel center or sample is checked if covered by a triangle: **inside-outside test**
- **Fragment** is generated for part of the pixel that overlaps the triangle
- **Fragment properties** are generated by **interpolating vertex data**
  - Due to perspective distortion, **perspective-correct interpolation** is needed
  - **Fragment properties include**: fragment depth and any shading data from geometry processing stage stored per vertex, e.g., color, normal, etc.
- All pixels, that is **fragments, inside primitive are sent to pixel processing stage.**

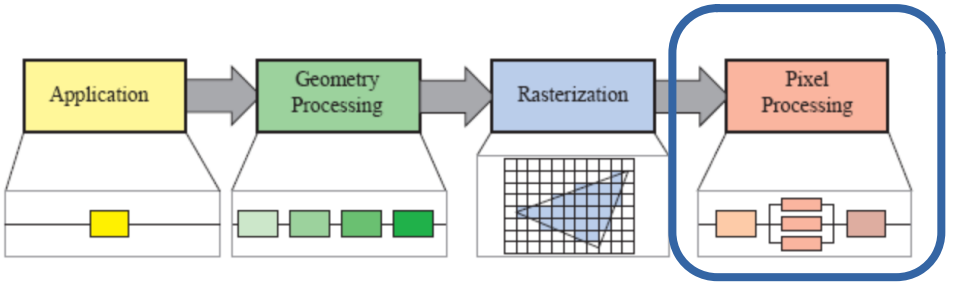


More info: https://people.csail.mit.edu/ericchan/bib/pdf/p15-mccormack.pdf

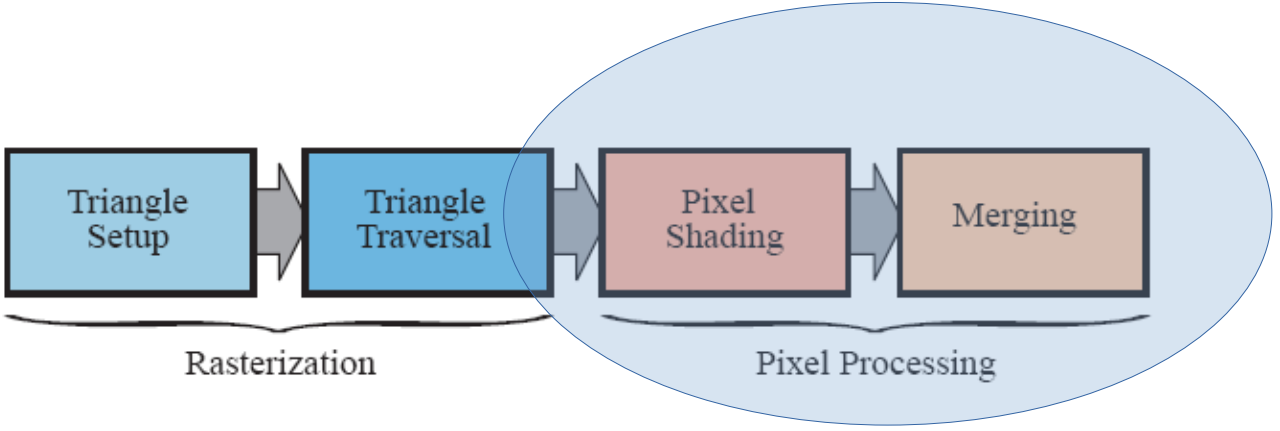
4. Pixel processing stage



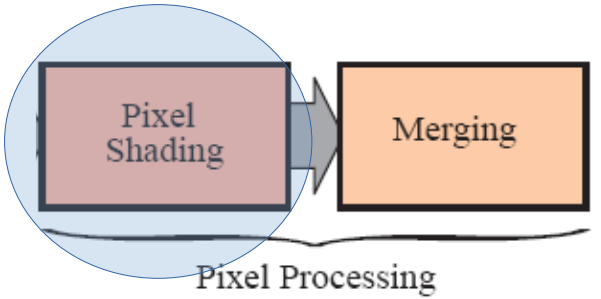
4. Pixel processing stage



- **Per-pixel or per-fragment computations** are done on pixels or samples (fragments) inside a primitive
  - Visibility solving using depth value
  - Shading: color computation
  - Blending: combining computed pixel colors into color buffer
- Pixel processing stage can be divided into:
  - **4.1 Pixel shading**
  - **4.2 Pixel merging**



4.1 Pixel shading



- **Per-pixel/fragment shading** is performed here
  - **Interpolated vertex attributes and shading data** from previous stage are used to compute color
- Pixel shading stage is performed by **programmable GPU cores**
  - Programmer supplies a **program for pixel (fragment) shader** that contains any desired computations
  - A **shader program** is executed per-pixel/per-fragment to determine its color
  - To compute object color, **shader program** defines **shading equations: e.g., direct illumination equation using scattering function** and **texture** (image or procedural)
- **Result of this stage is one or more colors for each pixel/fragment** that are passed further the pipeline

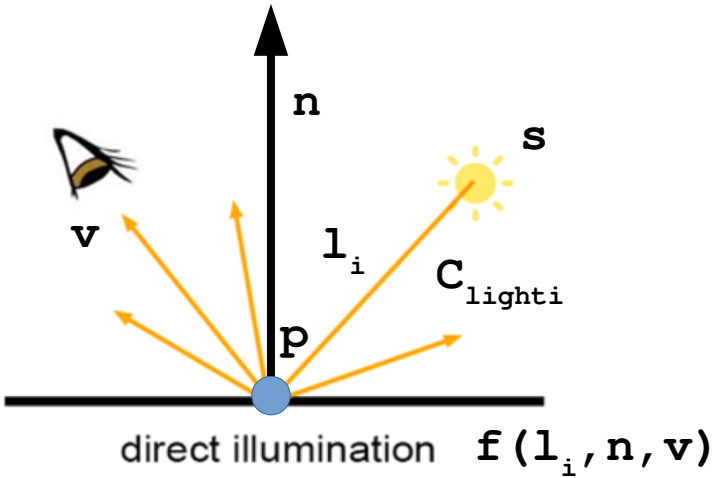
Pixel shading: computing pixel color

- Pixel/fragment shader use **shading equation** to compute the pixel color
- Shading equation often describes **direct illumination.** To compute color it uses:
  - Light position (**s**) and color (**c_{light}**)
  - Camera view direction (**v**)
  - Object material (BRDF, texture): `f(l_i, n, v)`
  - Object shape: normal (**n**)

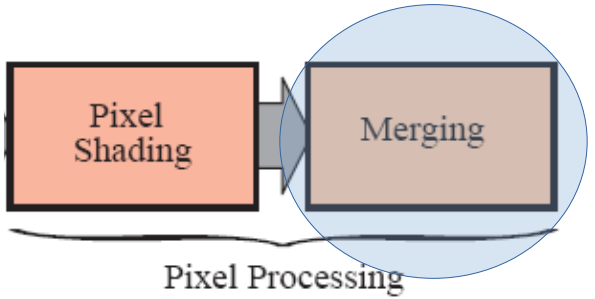
$$c_{shaded(p,v)} = \sum_{i=1}^{n} f(l_i, n, v) c_{light_i} (n \cdot l_i)^+$$

BRDF

Light source color



4.2 Pixel merging

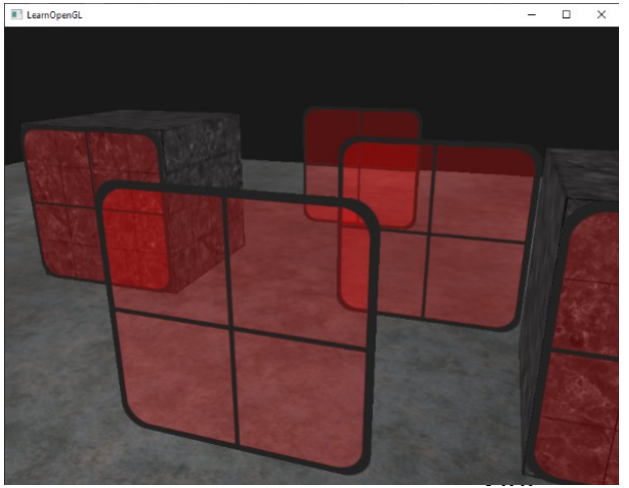


- Information for each pixel generated in pixel/fragment shader is stored in **color buffer**
  - Color buffer: **Rectangular array of of colors** `(r,g,b)` same size as image
- Task of merging stage is to **combine new fragment color produced by pixel shading stage with the color currently stored in color buffer**.
- This stage is not fully programmable, but highly configurable for achieving various effects (e.g., transparency).
  - Tendency is towards more programmability

DirectX calls this stage output merger. OpenGL calls this stage per-sample operations.

Pixel merging: color blending and visibility

- **Color blending**: **blending of new and current fragment color in color buffer**
  - For **opaque surfaces blending is not needed**: fragment color simply replace the previously stored color
  - For **transparent surfaces blending is needed:** new and old fragment color in color buffer must be combined
  - Blending can perform a **large number of operations involving color and alpha values** (useful for compositing)
    - Combinations of multiplication, addition, subtractions, min, max, bit-wise logic, etc.



https://learnopengl.com/Advanced-OpenGL/Blending

- **Resolving visibility: depth testing**
  - Color buffer should contain only color of triangles visible from camera point of view
  - Solved using **z-buffer/depth-buffer**

Pixel merging: frame buffers

- Pixel merging stage stage can use following buffers for computations:
  - **z-buffer**
  - **Alpha channel** (part of color buffer)
  - **Stencil buffer**
- **Frame buffer** generally consists of all buffers on a system.
- Different buffers that are generated can be used for **compositing** purposes.

z-buffer

- When the whole scene has been rendered, the **color buffer** should contain **colors of the primitives** (e.g., triangles) in the scene **visible from camera point of view**.
- For most graphics hardware this is achieved using, **z-buffer (depth buffer).**
  - Z-buffer has same **size and shape as color buffer**, but for each pixel it stores **z-value to the closest triangle**.
- **Depending on depth value of current triangle for current pixel and depth value for current pixel in z-buffer, color-buffer and z-buffer are updated:**

  1. When a triangle is being rendered at a certain pixel, z-value at that pixel is being computed and compared to the z-buffer
  2. If new z-value is smaller then value in z-buffer then that pixel is rendered closer to camera than previous pixel: color buffer and z-buffer must be updated with color and z-value of that pixel.
  3. Otherwise, color and z-buffer are not changed.

z-buffer example

- **Once all triangles have been processed, depth buffer contains distance to visible object from camera**
  - Gray scale image where each pixel represents depth from camera to triangle
  - Depth is useful for visibility solving, shading and post-processing (e.g., depth of field, fog, shadow mapping, etc.)

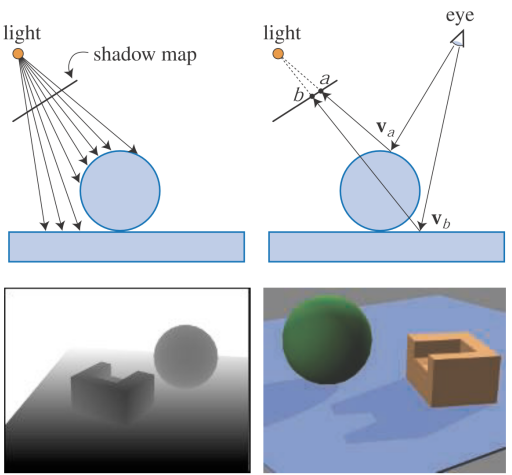
Color buffer.

z-buffer.

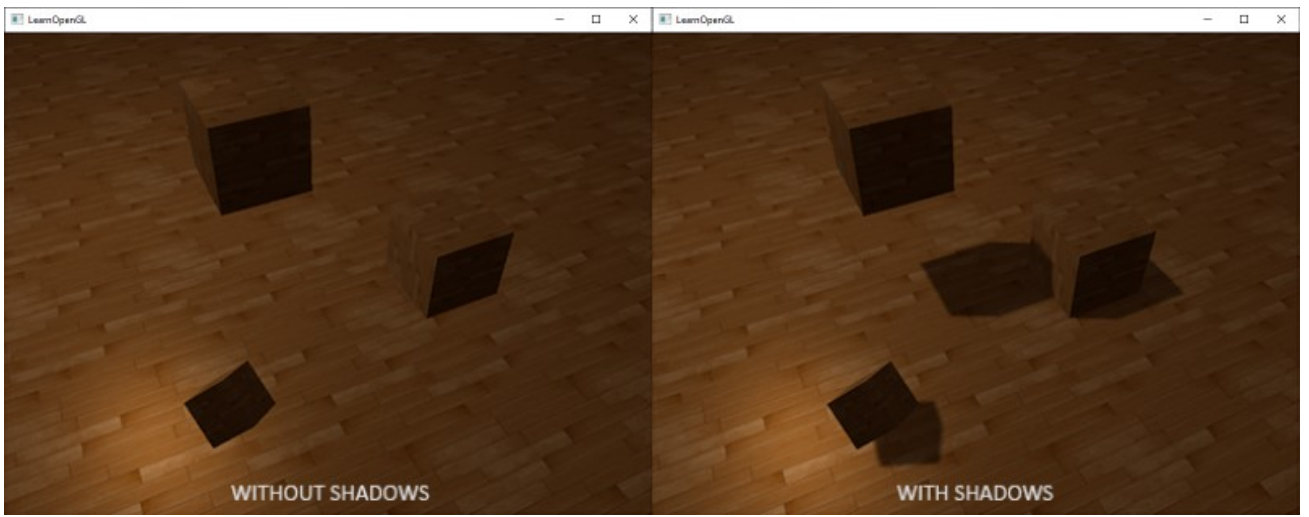
https://docs.chaos.com/display/VFBlender/Z-Depth

z-buffer example

- Z-buffer can be rendered not only from camera's point of view but also from light's point of view enabling shadow mapping technique



Rendering from light's point of view fills z-buffer with depth to closest visible objects to the light.



https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping

z-buffer properties

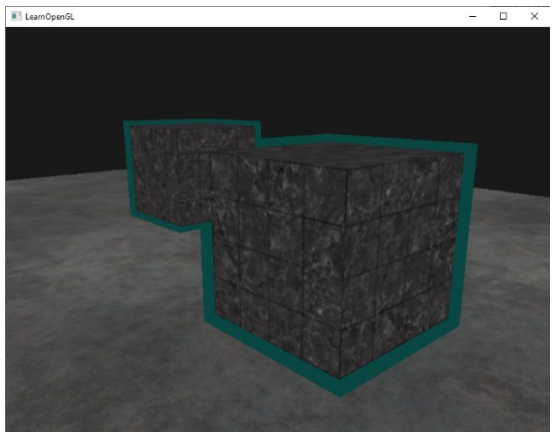
- **O(n)** complexity, where n is number of triangles
- **Order independent:** allows triangles to be rendered in any order.
- z-Buffer **stores only single depth value for each pixel**
  - **Can not be used for partially transparent objects**
- Partially transparent objects must be rendered after all opaque triangles using end to front order
- **early-z**: many GPUs perform some merge testing before pixel shader is exectued
  - z value of fragment is available after rasterization and it can be used for testing visibility and culling before pixel shader
  - Pixel shader can change z-depth of the fragment or discard whole fragment. In this case, early-z is not possible.

Alpha channel

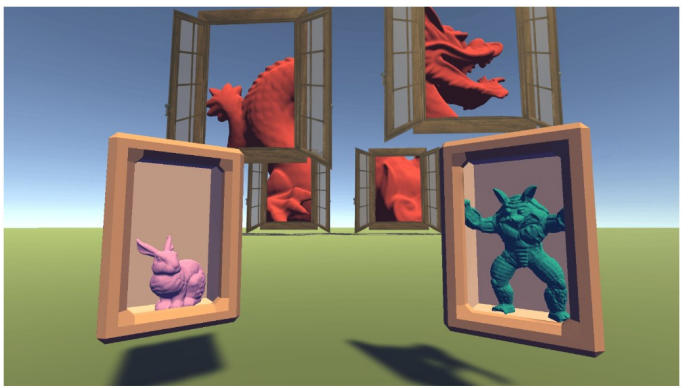
- Alpha channel is associated with color buffer and stores opacity, **alpha**, value for each pixel.
- Alpha value can be used for selective discarding of pixels using **alpha test feature** or pixel (fragment) shader.
  - This ensures that **fully transparent pixels/fragments do not affect the z-buffer.**

Stencil buffer

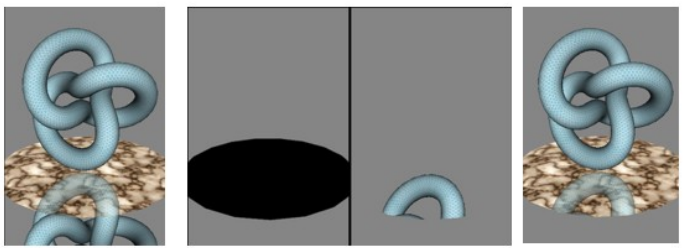
- Stencil buffer is **offscreen buffer** that **records locations of rendered primitives**
  - **Buffer contents are used to control rendering into color and z-buffer**
  - Example: filled rectangle is rendered into stencil buffer. This buffer is then used to render scene primitives into the color buffer only where the rectangle is present
- Stencil buffer is used in combination with different operators for **generating special effects**



Stencil buffer can be used to generate object outlines:
https://learnopengl.com/Advanced-OpenGL/Stencil-testing



Stencil buffer can be used to render in color buffer only on specific locations.
https://www.ronja-tutorials.com/post/022-stencil-buffers/



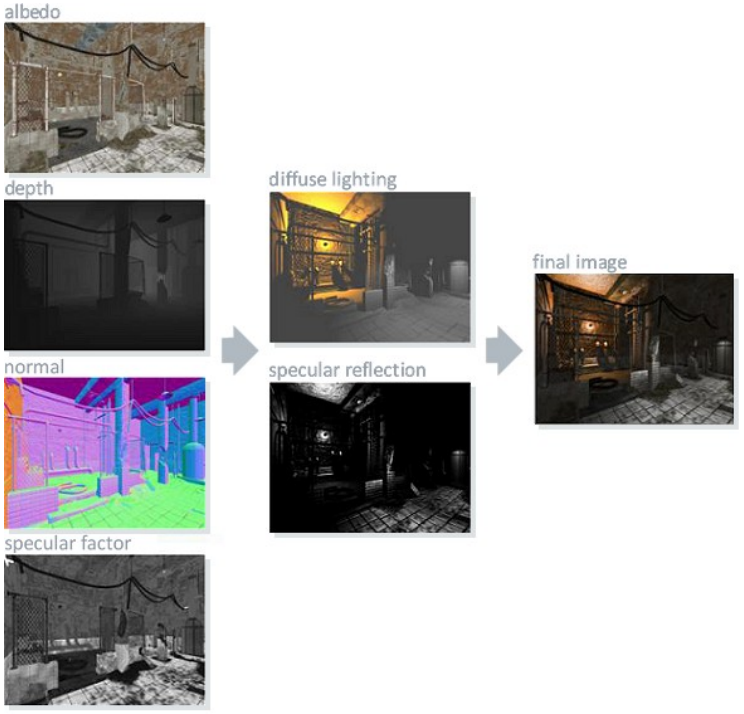
Problem: want to limit reflection to marbled floor

Stencil and stenciled reflection

result

Multiple render targets

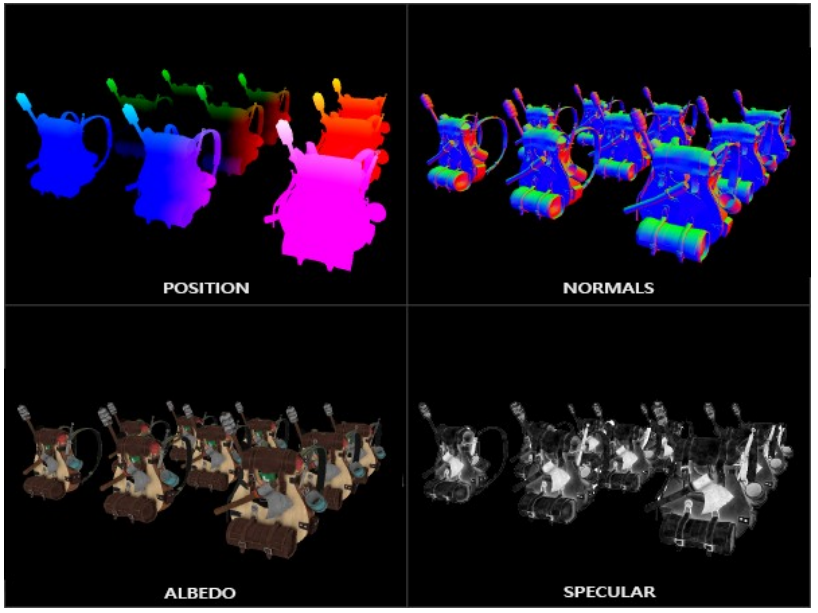
- Instead of storing results of pixel shader program directly to color buffer and z-buffer, multiple sets of values can be generated for each fragment and saved to different buffers each called **render targets**.
- **Single rendering pass can generate multiple rendering targets**:
  - Color image
  - Object identifies
  - Depth values
  - Surface normals
  - Etc.



https://stackoverflow.com/questions/6180259/point-to-multiple-render-targets

Forward vs deferred rendering/shading

- **Rendering pipeline discussed for now is called forward rendering/shading**
- **Multiple rendering targets inspired** different type of rendering pipeline called **deferred rendering (shading), a multi-pass pipeline:**
  - **First (geometry) pass resolves visibility** and stores data relevant for shading in buffers, e.g., object's location buffer, normals buffer, albedo buffer, material properties, z-buffer, etc. → **G (geometric) buffer**
  - **Successive (lighting/shading) passes** efficiently use G buffer information for shading
- Deferred shading is practical rendering method often used in production renderers

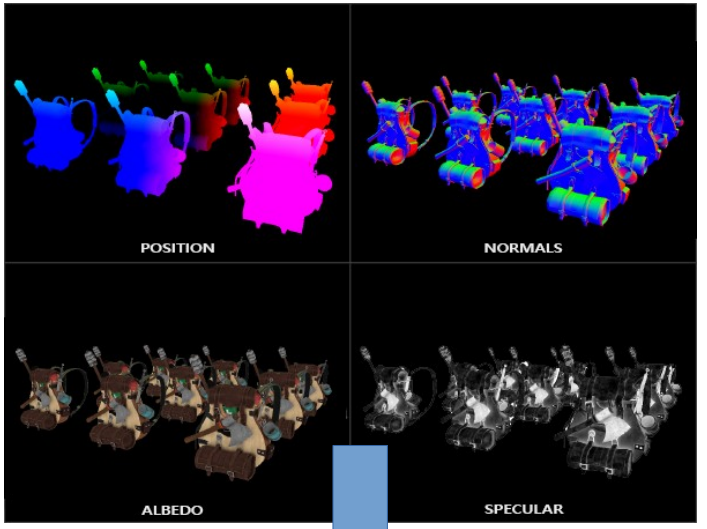


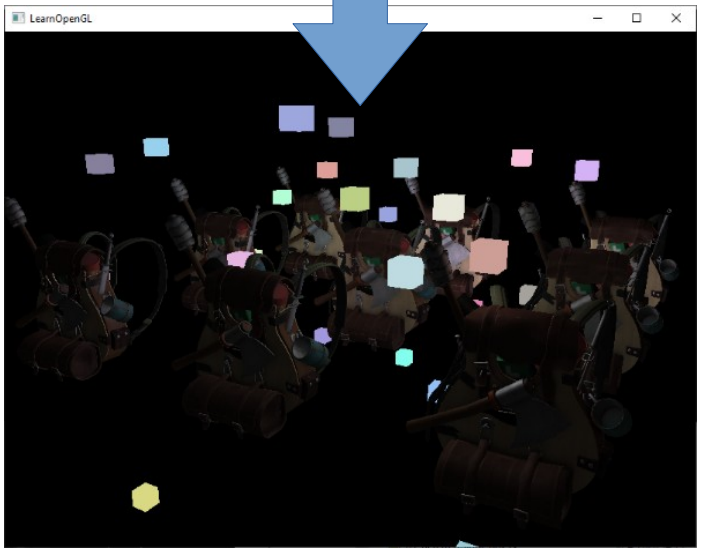
G-buffer is full-screen image (buffer) in which relevant data for shading is stored.

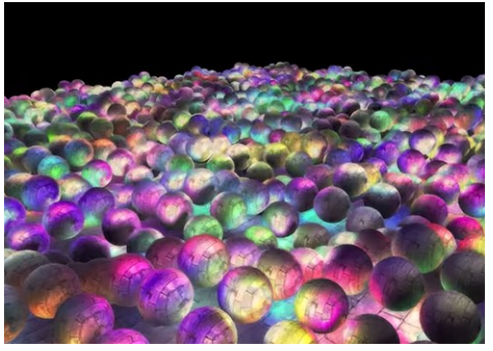
https://learnopengl.com/Advanced-Lighting/Deferred-Shading

Deferred shading

- With deferred shading, **shading is applied only actually visible fragments** G-buffer information.
- This way, **large number of lights with complex shading** performed in fragment shader can be computed more efficiently.
- **Problems:**
  - Since only one surface is stored per pixel, **basic deferred shading can not support transparency**
    - Combination of forward shading for transparent and deferred shading for opaque objects can be used
  - **Anti-aliasing is memory expensive**
    - Detecting edges with image processing tools makes anti-aliasing more tractable since only those regions require more samples per pixel







Reading: https://www.beyond3d.com/content/articles/19/

Displaying pipeline output

- Triangles that have reached and passed rasterizer stage are visible from camera point of view and their colors will be stored in color buffer.
- Display device (screen) will show color buffer after all operations are done.
- **Since this takes time, double buffering method** is used:
  - Rendering is performed in a **back buffer**.
  - Contents of back buffer are swapped with the contents of **front buffer –** buffer which is shown on display device.
  - Swapping occurs during **vertical trace** – a time when it is safe to do so.

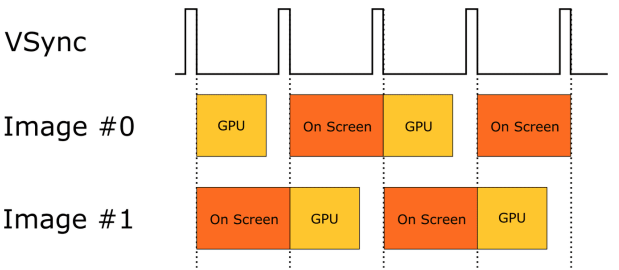


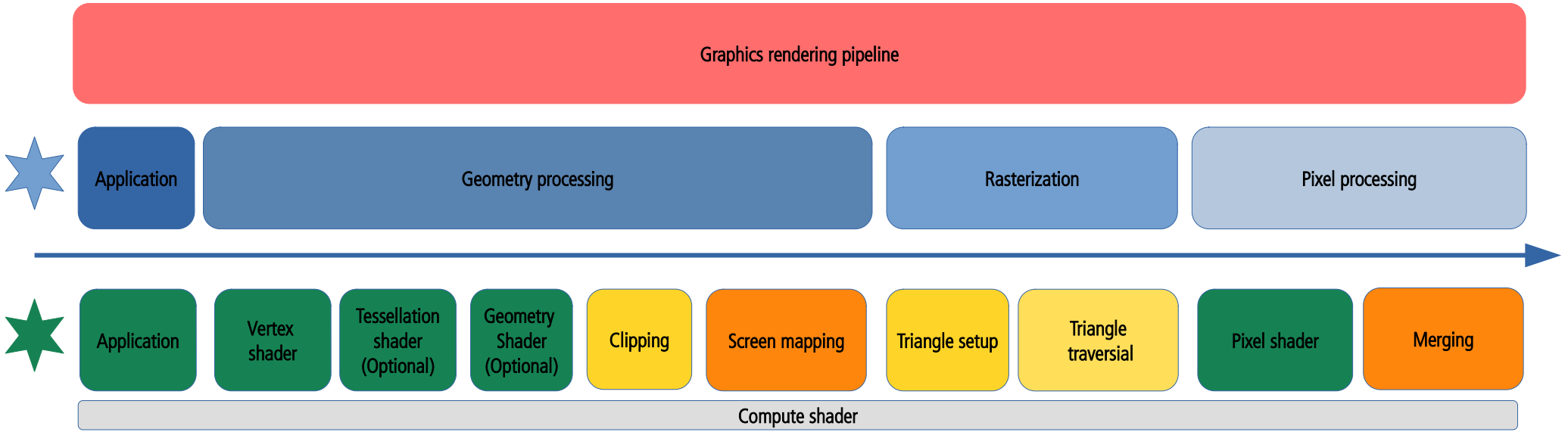
Figure 4: Correct double buffering behavior

https://developer.samsung.com/sdp/blog/en-us/2019/07/26/vulkan-mobile-best-practice-how-to-configure-your-vulkan-swapchain

Logical GPU model

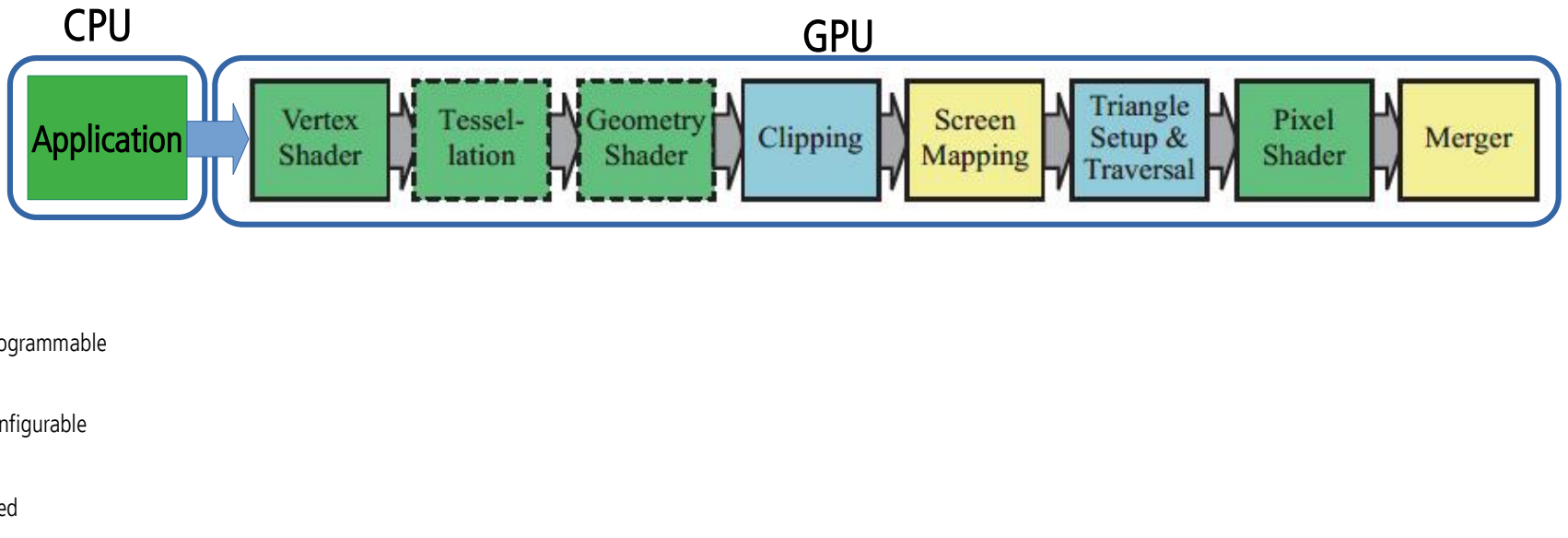
Programming API

Graphics rendering pipeline: conceptual vs logical (API) model



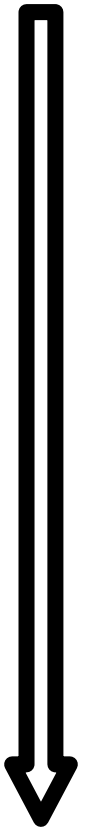
Logical GPU model: API

- **Logical GPU rendering model is exposed to programmer as API**
- **Underlying hardware implementation** of **conceptual graphics pipeline** varies by hardware vendor (e.g., Nvidia, AMD, etc.)



Graphics stages

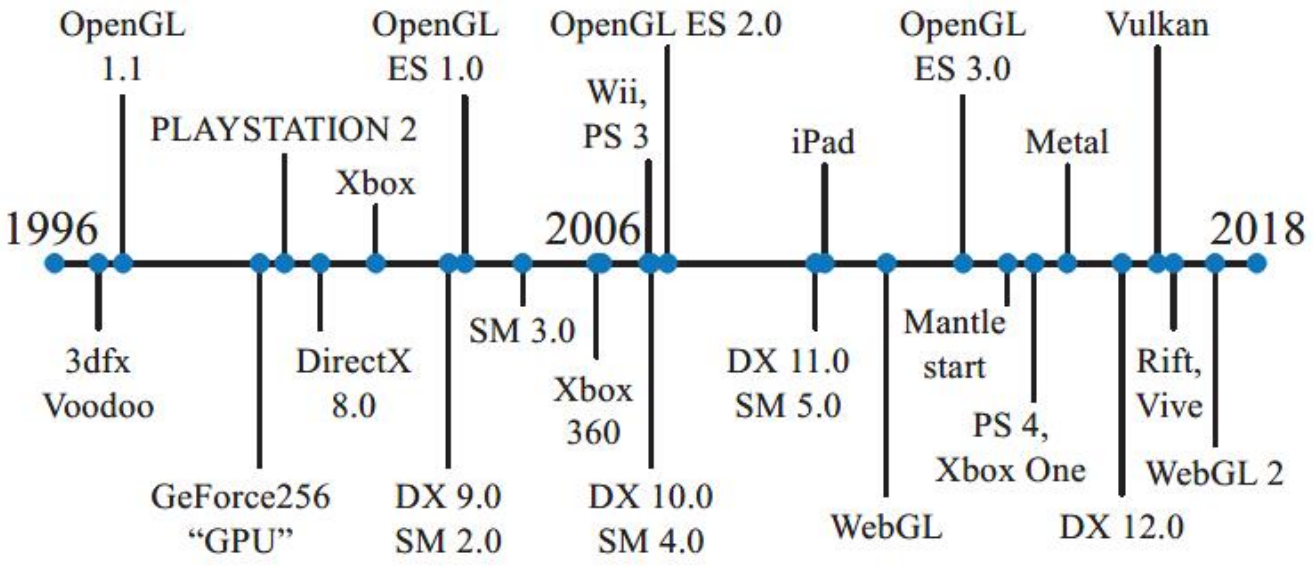
- **Application stage**
  - **Specification of 3D scene**: objects (geometry, material, transformations ,etc.), lights (color, transformations, etc.), cameras (transformations, focal length, field of view, etc.)
  - **Specification of shader program** (e.g., vertex and fragment shaders)
  - **Configuration of graphics pipeline state** (e.g., depth testing)
  - **Draw (rendering) commands** (i.e., sending information to GPU for rendering)
- **Driver:**
  - **Translation of API instructions to hardware operations**
  - **Compilation of shader programs**
- **Graphics hardware (GPU)**
  - **Geometry processing, rasterization, pixel processing → shaders**
  - **Visibility resolving, clipping, shading, etc.**



Programming API

- **Application stage (CPU)** can be written in various languages: C, C++, Python, Rust, etc.
- **Graphics API for accessing GPU hardware:**
  - OpenGL (cross-platform), https://www.khronos.org/
    - WebGL (browser-based graphics)
    - OpenGL ES (embedded systems)
  - Vulkan (cross-platform), https://www.khronos.org/
  - DirectX (Windows), https://devblogs.microsoft.com/directx/
  - Metal (iOS), https://developer.apple.com/metal/
- **Focus: OpenGL**

Evolution of graphics API and hardware



https://www.realtimerendering.com/

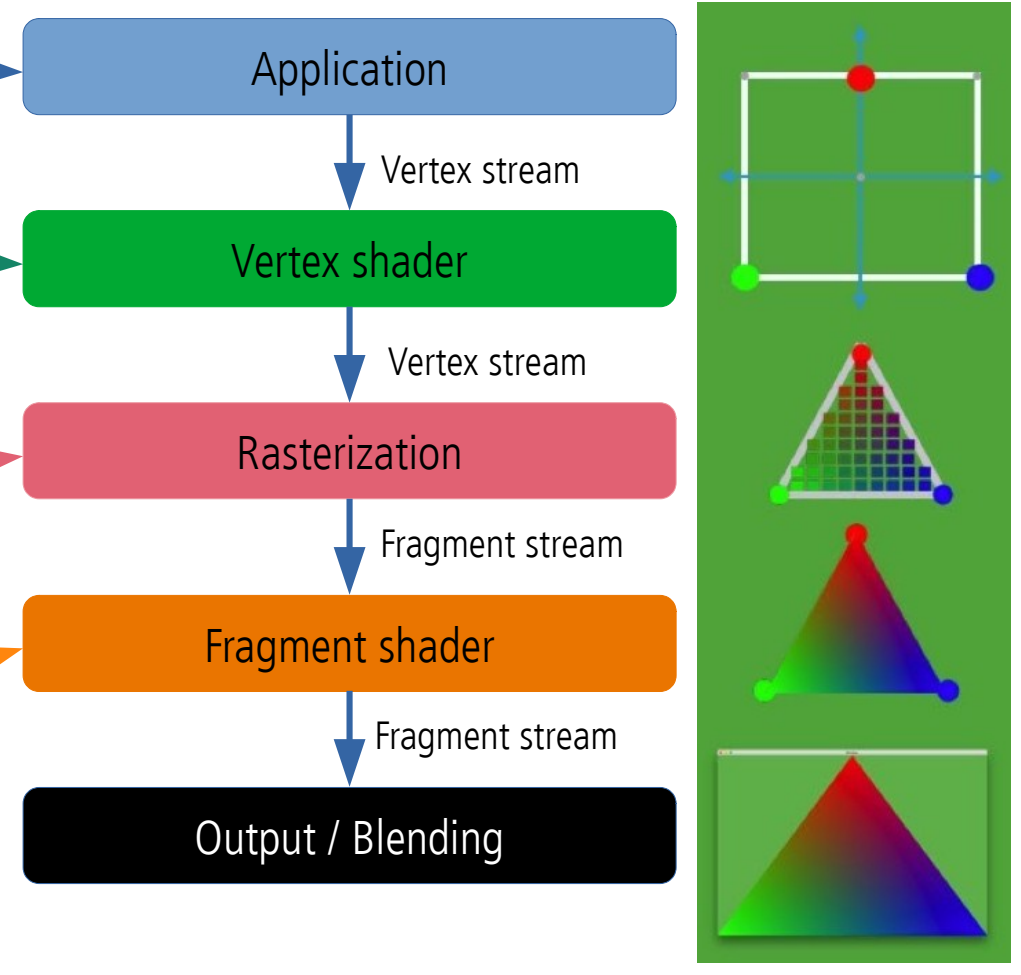
Logical GPU stages

Application
- Definition of a 3D scene
- Triangle vertices and vertex attributes

Geometry processing
- Transformation and projection of triangulated mesh vertices
- Object → world → camera → screen → clip coordinates

Rasterization
- Resolving visibility
- Generating fragment for each pixel covering triangle

Pixel/Fragment processing
- Shading
- Light and color computation



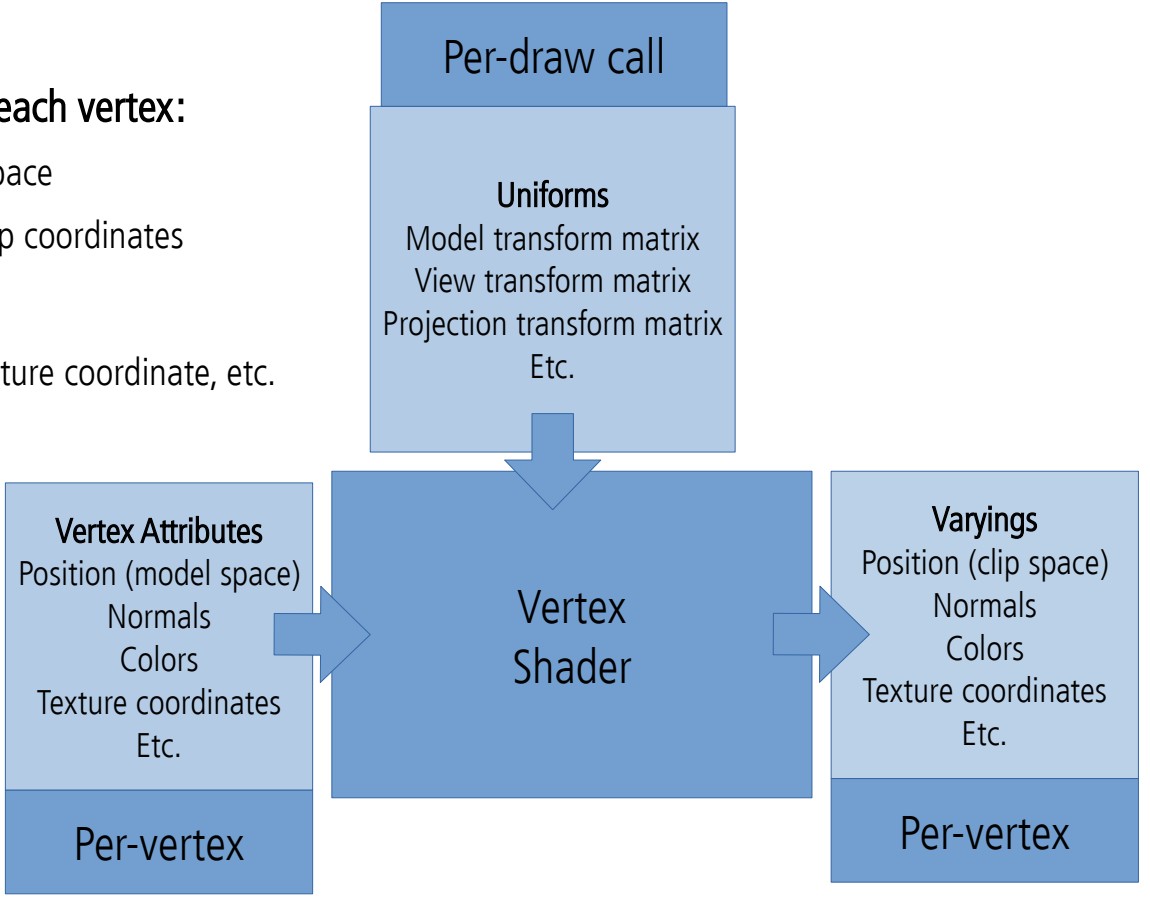
https://www.youtube.com/watch?v=F7bpcyPhiH8&ab_channel=2etime

Shader programming

- **Modern graphics** programming relies on programmable vertex and fragment shaders for rendering
  - **Unified shader design**: vertex, pixel, geometry, tessellation shaders share common programming model – same instruction set arhitecture (ISA)
  - GPU with cores which supports unified shader design: **unified shader architecture**
- **Shading languages:**
  - **GLSL (OpenGL, Vulkan)**
  - HLSL (DirectX)
- **Life cycle:**
  - Written and compiled on application stage
  - Linked at shader stages
  - Run when rendering objects

Vertex shader

- **Responsible for geometry processing, executes per each vertex:**
  - Transformation of mesh vertices from model to world space
  - Transformation of mesh vertices from world space to clip coordinates
- **Inputs are specificed by application:**
  - **Vertex attributes**: vertex coordinates, normals, color, texture coordinate, etc.
  - **Constants** – **uniforms**: e.g., transformation matrices
- **Outputs:** vertex variables – **varyings**:
  - Position in clip space (`gl_Position`)
  - Vertex variables (e.g., colors, normals, etc.)



Vertex shader program

```
#version 330 core

// Input (vertex attributes).
layout (location = 0) in vec3 vertexPosition;
layout (location = 1) in vec3 vertexNormal;

// Output (varyings).
out vec3 normal;

// Uniforms (constant parameters).
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    // Compute transformation of vertex coordinates
    // from world to clip space.
    gl_Position = projection * view * model * vec4(vertexPosition, 1.0);

    normal = mat3(transpose(inverse(model))) * vertexNormal;
}

```

https://learnopengl.com/Lighting/Basic-Lighting

Application stage: preparing uniforms for vertex shader program

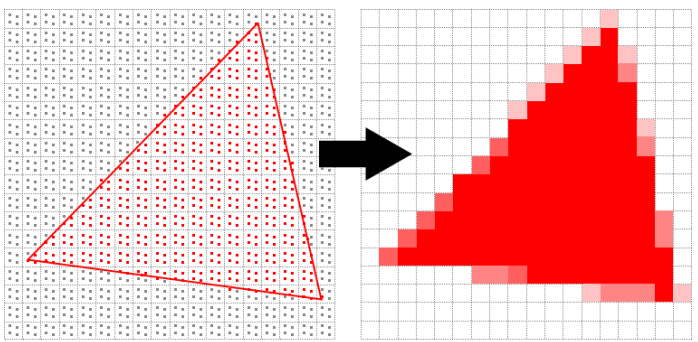
```
1
2   // Compile vertex shader program, link program.
3   GLint vertexShaderProgram = buildShaderProgram( ... );
4
5   // Create transformation matrix.
6   GLfloat* modelMatrix = createModelMatrix( ... );
7   GLfloat* viewMatrix = createViewMatrix( ... );
8   GLfloat* projectionMatrix = createProjectionMatrix( ... );
9
10  // Use shader program for draw calls.
11  glUseProgram( vertexShaderProgram );
12
13  // Obtain index of uniform in shader program.
14  GLint modelMatrixLocation = glGetUniformLocation( vertexShaderProgram, "model" );
15  GLint viewMatrixLocation = glGetUniformLocation( vertexShaderProgram, "view" );
16  GLint projectionMatrixLocation = glGetUniformLocation( vertexShaderProgram, "projection" );
17
18  // Set model matrix uniform.
19  glUniformMatrix4fv( modelMatrixLocation, GL_FALSE, modelMatrix );
20  glUniformMatrix4fv( viewMatrixLocation, GL_FALSE, viewMatrix );
21  glUniformMatrix4fv( projectionMatrixLocation, GL_FALSE, projectionMatrix );
22
```

Application stage: preparing geometry for vertex shader program

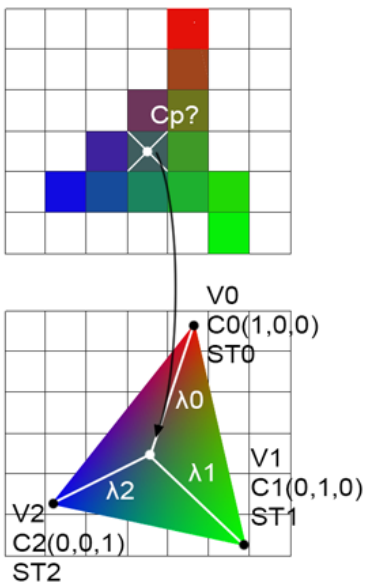
```
1
2   // 3 vertices of a triangle.
3   float vertices[] =
4   {
5       0.0, 0.0, 0.0,
6       1.0, 0.0, 0.0,
7       0.0, 1.0, 0.0
8   };
9   // Create buffer in GPU memory.
10  GLint vertexBuffer;
11  glGenBuffers( 1, &vertexBuffer );
12  // Bind vertexBuffer.
13  glBindBuffer( GL_ARRAY_BUFFER, vertexBuffer );
14  // Upload vertices to buffer.
15  glBufferData (GL_ARRAY_BUFFER, vertices, GL_STATIC_DRAW );
16  // Get vertex position in vertex shader.
17  GLint vertexIndex = glGetAttribLocation ( vertexShaderProgram, "vertexPosition" );
18  // Associate buffer with shader attribute and set interpretation.
19  glVertexAttribPointer( vertexIndex, GL_FLOAT, 3, false, 0, 0 );
20  // Enable vertex attribute.
21  glEnableVertexArray( vertexIndex );
22
```

Rasterization

- Given transformed and projected vertices (with associated vertex attributes data for shading) the goal is to **find all pixels/fragments which are inside the triangle** to be used in fragment shader.
- For each pixel, multiple samples (points) are created and tested if are inside triangle → **fragments generation**
- **Rasterization outputs fragments with:**
  - **Pixel coordinates**
  - **Interpolated varyings (vertex attributes) using barycentric interpolation**
  - **Interpolated depth value**

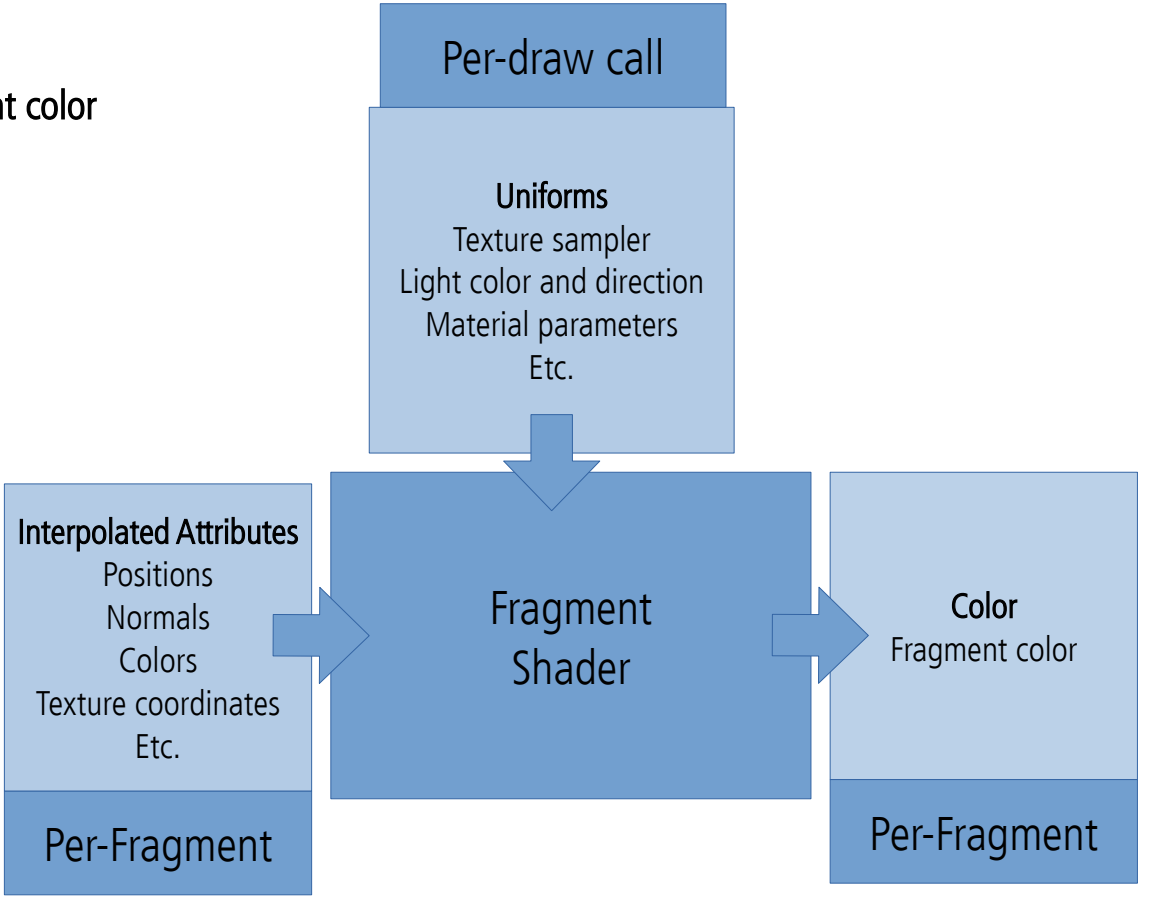


Using 4 samples per pixel (MSAAx4)



Fragment shader

- Responsible for shading, that is, computation of fragment color which is then combined into pixel color of color buffer
- Fragment shader **input**:
  - Interpolated vertex attributes from vertex shader
  - Texture samplers (texture image)
  - Uniforms (constant parameters)
- Fragment shader output:
  - Fragment color

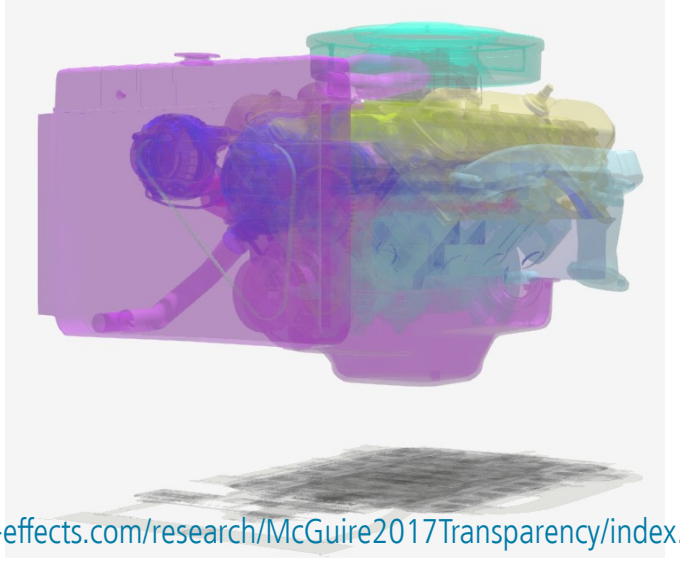


Fragment shader program

```
1
2   #version 330 core
3
4   // Fragment shader output.
5   out vec4 fragColor;
6
7   // Input (interpolated from vertex shader output).
8   in vec3 normal;
9
10  // Uniforms, constant input.
11  uniform vec3 lightDirection;
12  uniform vec3 lightColor;
13  uniform vec3 objectColor;
14
15  void main()
16  {
17      vec3 nnormal = normalize( normal );
18      float diffuse = max( dot(nnormal, lightDirection), 0.0 );
19      FragColor = vec4( diffuse * objectColor * lightColor , 1.0 );
20  }
```

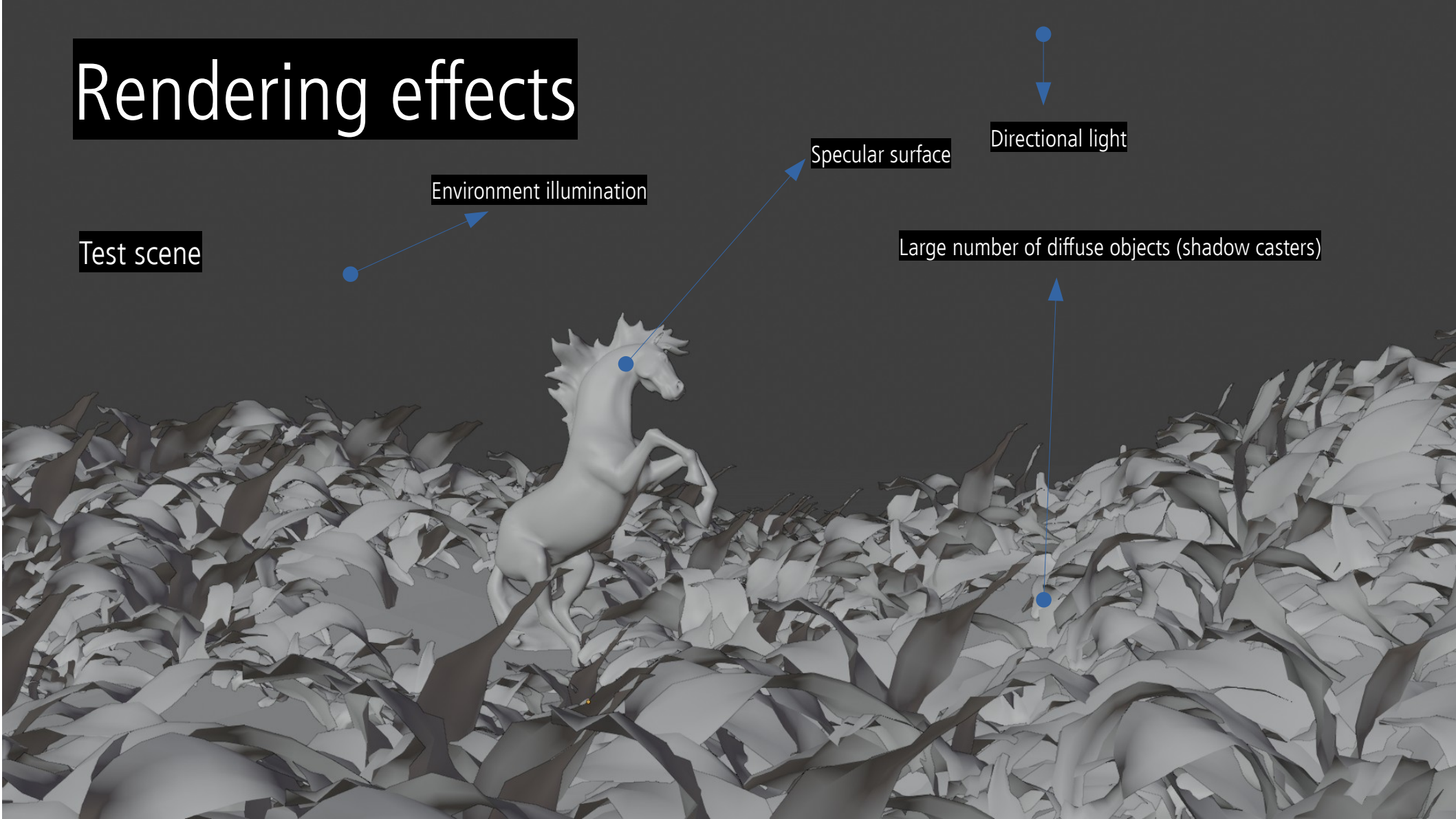
Output and blending

- Resolving visibility
  - Color buffer should contain only color of triangles visible from camera point of view
  - Performed using **z-buffer/depth-buffer**
- Blending new fragment color and existing pixel color in color buffer
- Alpha value testing
- Stencil testing



https://casual-effects.com/research/McGuire2017Transparency/index.html

Rendering effects




Ray-tracing based rendering (Blender, Cycles):
• Direct + indirect illumination
• Soft shadows
• Environment illumination

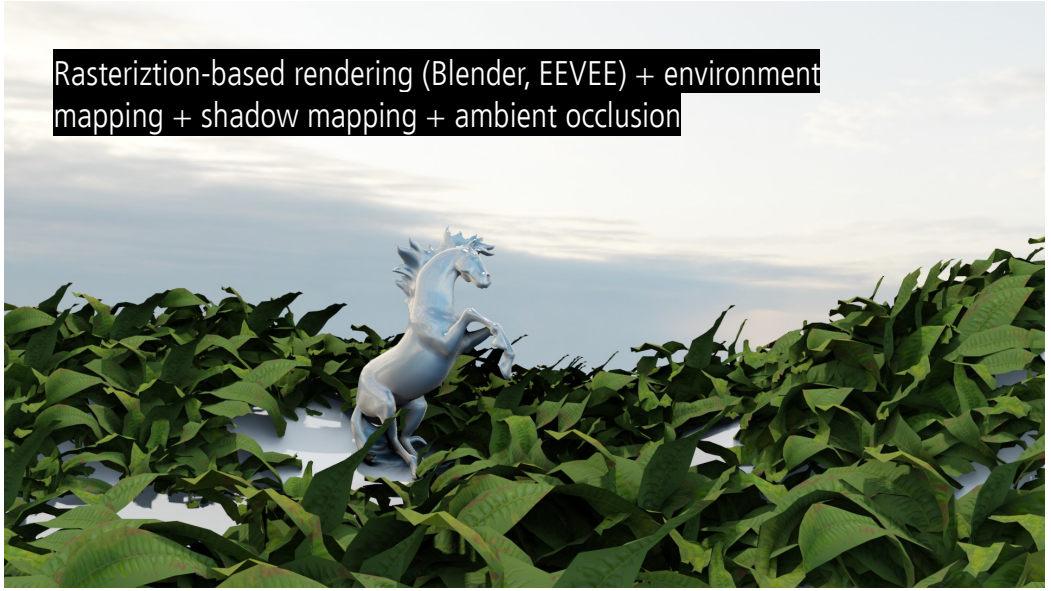
Basic rasterization-based rendering (Blender, EEVEE):
Direct illumination


Basic rasterization-based rendering (Blender, EEVEE):
• Direct illumination
• Environment mapping → environment illumination


Basic rasterization-based rendering (Blender, EEVEE):
• Direct illumination
• Environment mapping → environment illumination
• Shadow mapping → shadows


Basic rasterization-based rendering (Blender, EEVEE):
• Direct illumination
• Environment mapping → environment illumination
• Shadow mapping → shadows
• Ambient occlusion → more shadows

Rendering effects

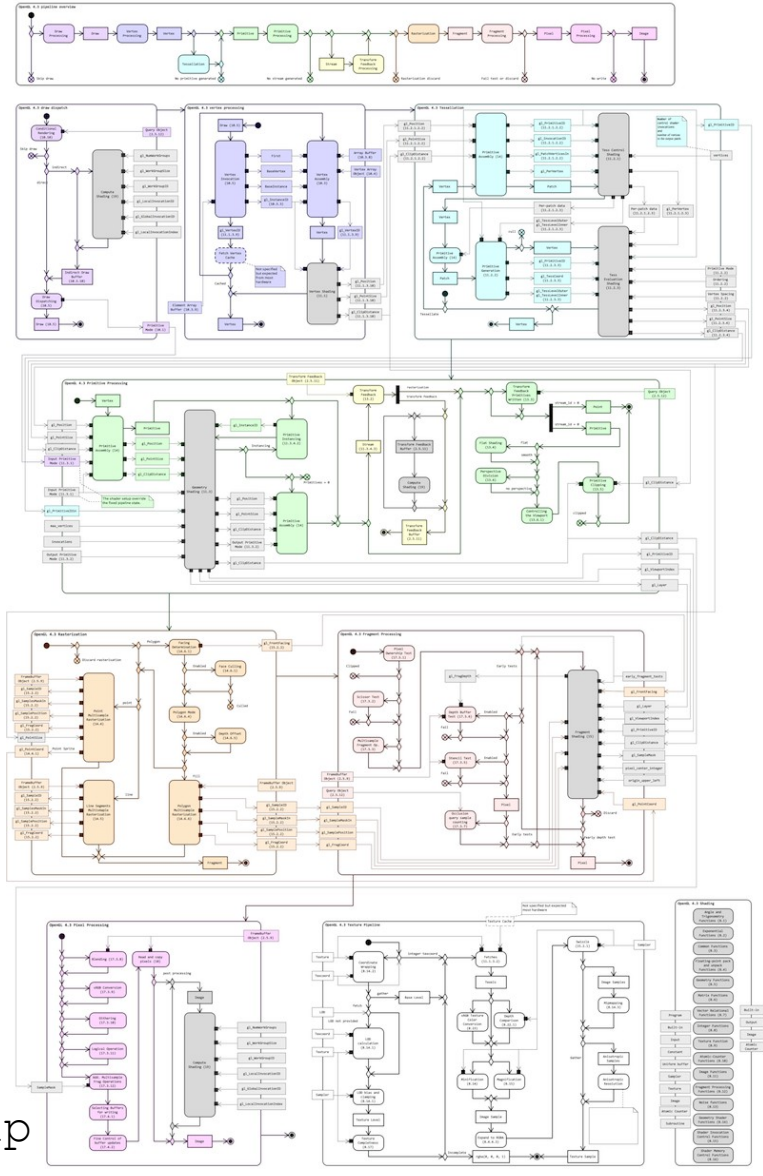


To render advanced light phenomena which can be elegantly achieved in ray-tracing, rasterization-based rendering requires additional work performed by vertex and fragment shaders:

- Shadow mapping
- Ambient occlusion
- Environment mapping

More into topic

- Scratchapixel:
  - https://www.scratchapixel.com/lessons/3d-basic-rendering/computing-pixel-coordinates-of-3d-point/perspective-projection.html
  - https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/overview-rasterization-algorithm.html
  - https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/projection-matrix-introduction.html
- OpenGL:
  - https://graphicscompendium.com/opengl/
  - https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview
  - https://learnopengl.com/Lighting/Basic-Lighting
- Graphics pipeline in general:
  - https://alaingalvan.gitbook.io/a-trip-through-the-graphics-pipeline/
  - https://www.g-truc.net/post-0580.html
  - https://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/
- Vulkan: https://vulkan-tutorial.com/Introduction
- More links: https://www.realtimerendering.com/



OpenGL 4.3. Pipeline map

Summary questions

- https://github.com/lorentzo/IntroductionToComputerGraphics/tree/main/lectures/13_rendering_rasterization

Repository

- https://github.com/lorentzo/IntroductionToComputerGraphics