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-based rendering
  - Rasterization: the core
  - Graphics rendering pipeline
  - Logical GPU model/API

Rasterization

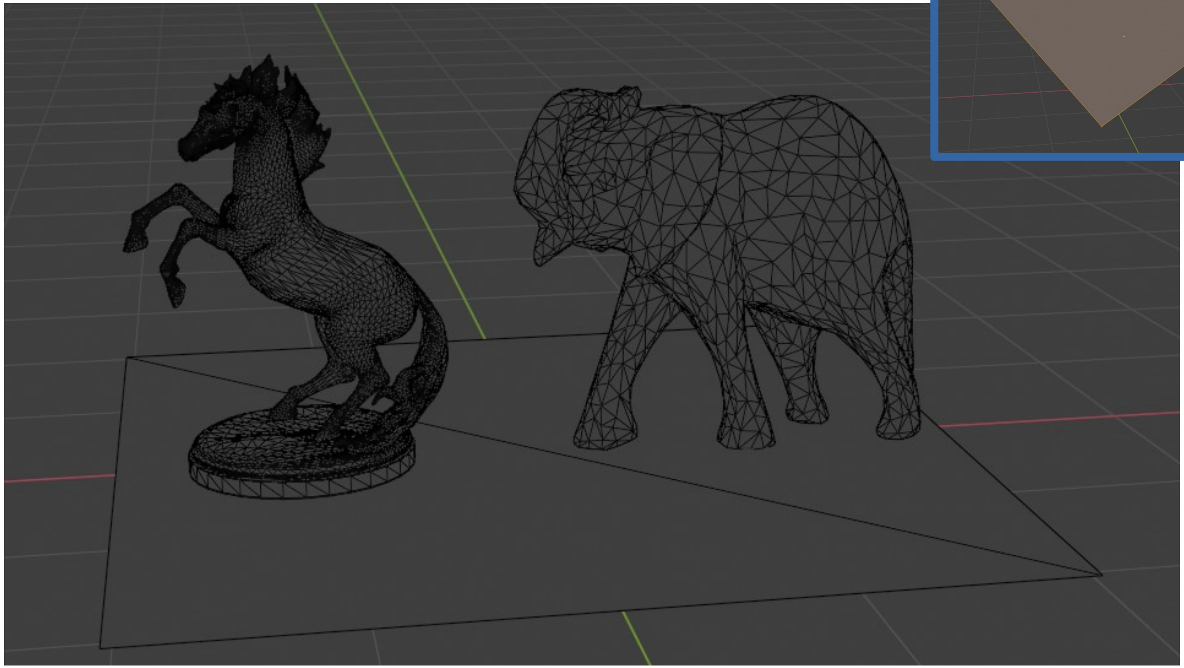
The core for visibility solving

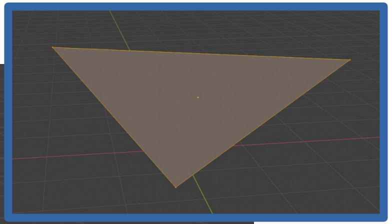
Rendering

- **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

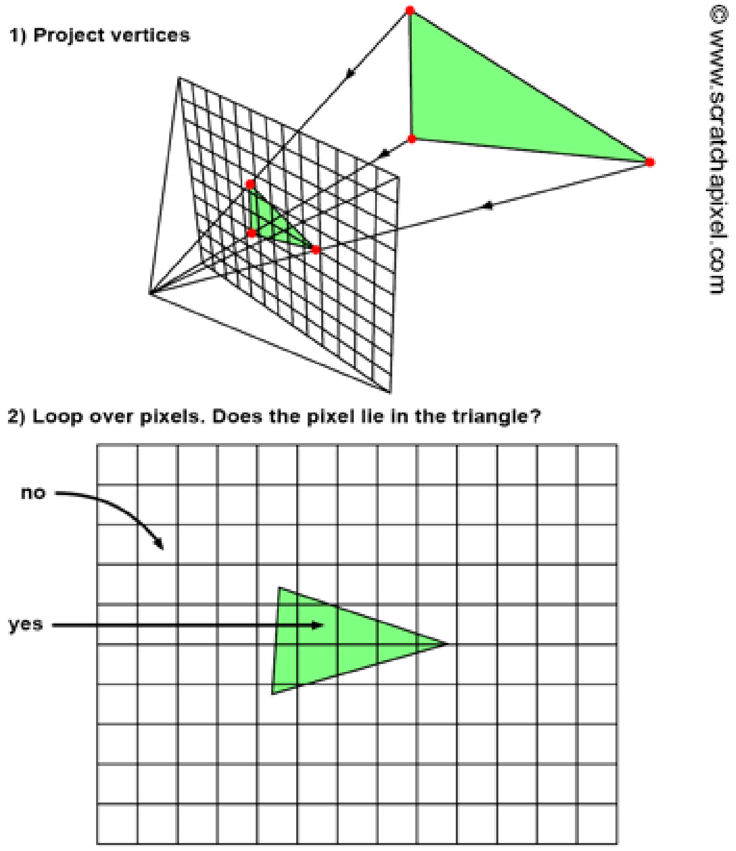
- **Visibility is geometrical problem using object shape representation** to determine which points in 3D scene are visible to each other
- **Triangulated mesh is shape representation used for efficient visibility solving**
  - Different shape representations can be converted to triangulated mesh using **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(v0,v1,v2,pixel.x, pixel.y):
            image(pixel.x, pixel.y) = triangle.color
```

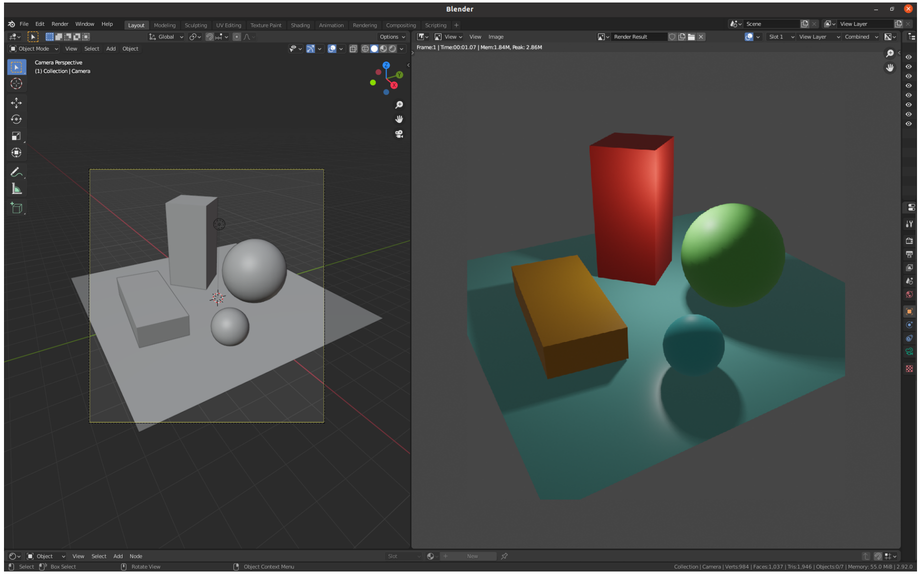
Projection step

Rasterization step

- Object centric approach: first loops over triangles, then loops over image pixels
- Note: looping over all triangles and all pixels is naive approach: further optimizations are introduced

Rasterization: buffers

- **Rasterization**: decomposing triangles into pixels forming raster image
  - Formally: rasterization of triangles into an image or frame buffer
- **During rasterization rendering, pixel information is stored in buffers** → 2D array of pixels which has same size as final image

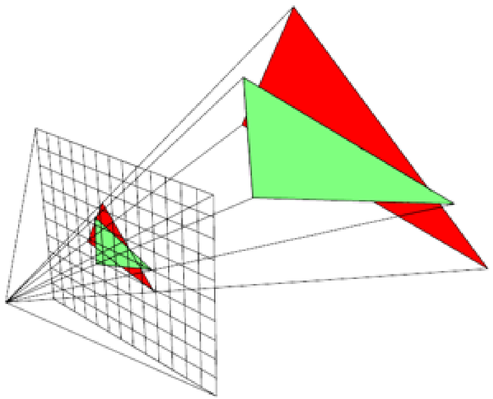


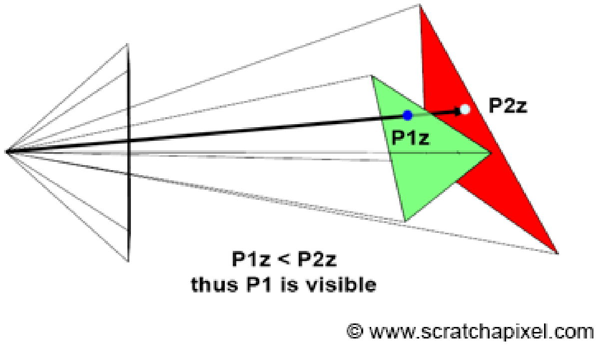
Frame buffer is 2D array, an image, containing the result of rendering ready for display

- Before rendering, frame-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 frame-buffer at that pixel location
- After rendering, when all triangles are rasterized and pixels are processed, the frame-buffer will contain the final image of the scene

Rasterization: z-buffer

- **Multiple triangles may overlap the same pixel in image**
- To correctly solve visibility, rasterization must find the first visible triangle
- This is done using **z-buffer or depth buffer**
  - 2D array with same dimensions as image but instead of colors it contains array of floating point numbers
  - z-buffer stores the distance of each pixel to the nearest object in the scene
- **Visibility computation with z-buffer:**
  - 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 and frame-buffer with color of this triangle at this pixel position
- Alternative: Painter's algorithm





Rasterization: frame-buffer and z-buffer

```
color frame_buffer = new float[imageWidth, imageHeight];
float z_buffer = new float[imageWidth, imageHeight];
for (int i = 0; i < imageWidth * imageHeight; ++i)
    frame_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;
    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;
            frame_buffer(pixel.x, pixel.y) = triangle.color
```

Projection stage

Rasterization stage

Rasterization: projection stage

- **Goal: Project triangles onto image plane (screen) and convert their coordinates from screen to raster space**
  - When triangles are converted to same space as pixels it is possible to compare their coordinates which is needed to find which pixels are overlapping triangle
  - This way, whole triangle can be rendered knowing only vertex positions
- Projection stage is responsible for **transforming object 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**

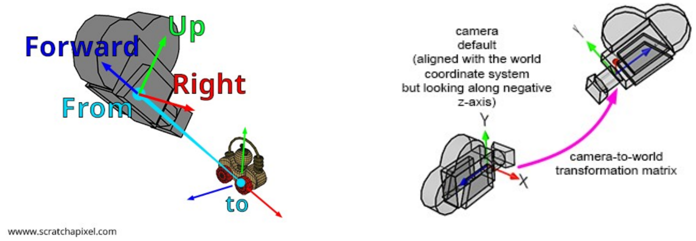
Projection stage: (1) world to camera space

World space to camera space:

- Triangulated mesh vertex coordinates are defined in world space $P_{world}(x, y, z)$
- Convert vertex coordinates $P_{world}(x, y, z)$ from world to camera space $P_{camera}(x, y, z)$ using **world-to-camera matrix**
- World-to-camera transformation matrix can be constructed using look-at notation

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

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




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

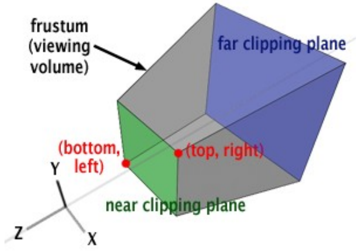
Reading:
https://www.scratchapixel.com/lessons/3d-basic-rendering/computing-pixel-coordinates-of-3d-point/mathematics-computing-2d-coordinates-of-3d-points.html

Projection stage: (2) camera to screen space

Camera space to screen (image plane) space: convert camera space vertex coordinates $P_{camera}(x,y,z)$ to screen (image plane) coordinates $P_{screen}(x,y,z)$

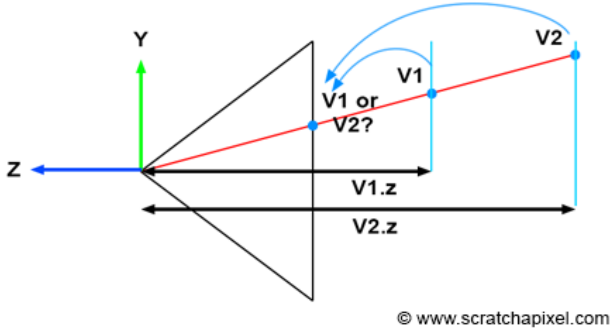
$$Pscreen.x = \frac{near * Pcamera.x}{-Pcamera.z}$$
$$Pscreen.y = \frac{near * Pcamera.y}{-Pcamera.z}$$
$$Pscreen.z = -Pcamera.z$$

- 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 distance from camera** → this will be **needed for resolving visibility and overlapping triangles**



If multiple points are projected on same point on screen, use z coordinate to solve visibility

Projection stage: (3) screen to NDC space

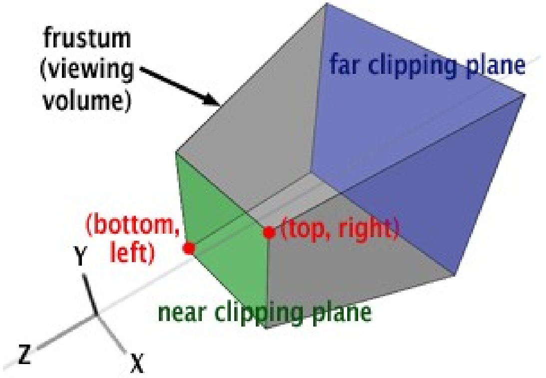
- NDC space coordinates are in range:
  - `[left,right]` → `[l,r]` for x coordinate
  - `[bottom, top]` → `[b,t]` for y coordinate
- GPU and real-time rendering standard (e.g., OpenGL, DirectX): NDC is `[-1,1]` for both coordinates
- **`(x,y)` coordinates from screen space are converted NDC space using:**

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

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

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

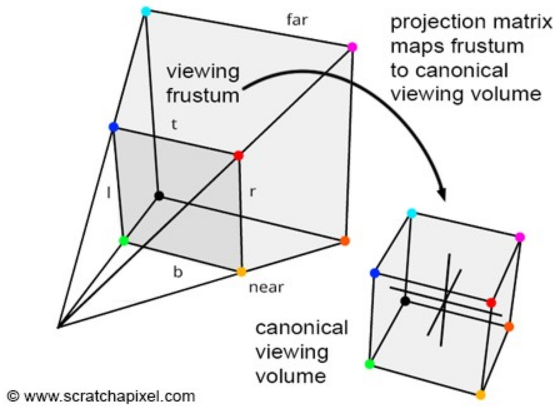
- **Z coordinate is remapped to [0,1] or [-1,1]**
  - If P lies on near clipping plane then remap to 0 (or -1)
  - If P lies on far clipping plane then remap to 1



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

Note: Screen and NDC space

- **Conversion of vertex coordinates from camera space to NDC space can also be done using projection matrix**, it replaces:
  - Perspective divide operation
  - Remapping from screen to NDC space
- **Projection matrix remaps viewing frustum into unit cube** – canonical view volume → **normalization of viewing frustum**
  - Working with unit cube as frustum makes computation much more tractable
- **GPU rendering uses projection matrix to convert vertex coordinates from screen space to clip space** and then to NDC and raster space
  - **Clipping**: triangles overlapping viewing frustum planes are "chopped off" (Cohen-Sutherland algorithm)



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

Perspective projection matrix is built using far (f), near (n), right (r), left(l), top (t), bottom (b) 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

Projection stage: (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)`

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

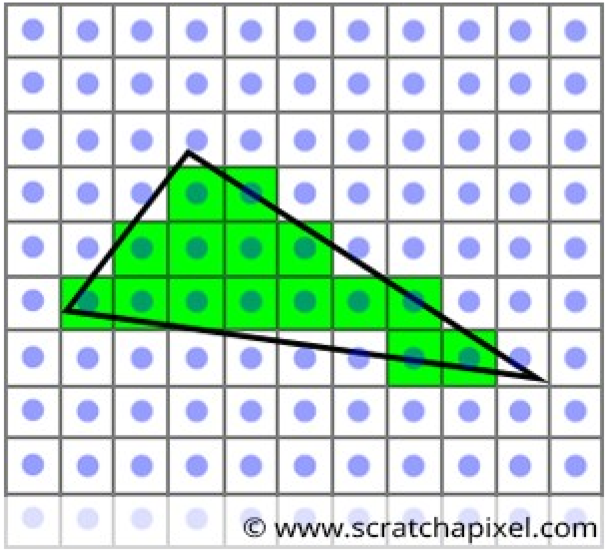
$$P_{raster}.y = 1 - \frac{P_{NDC}.y}{2} imageHeight$$

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

Triangle rasterization

- **Triangles are now converted to same space as pixels it is possible to compare their coordinates which is needed to find which pixels are overlapping triangle**
  - **This way, whole triangle can be rendered knowing only vertex positions**
- **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: scanline rasterization → based on Bresenham algorithm for drawing lines
  - For each pixel that overlaps triangle, compute:
    - **Barycentric coordinates**
    - **Depth**
    - Etc.

→ Needed for visibility and shading



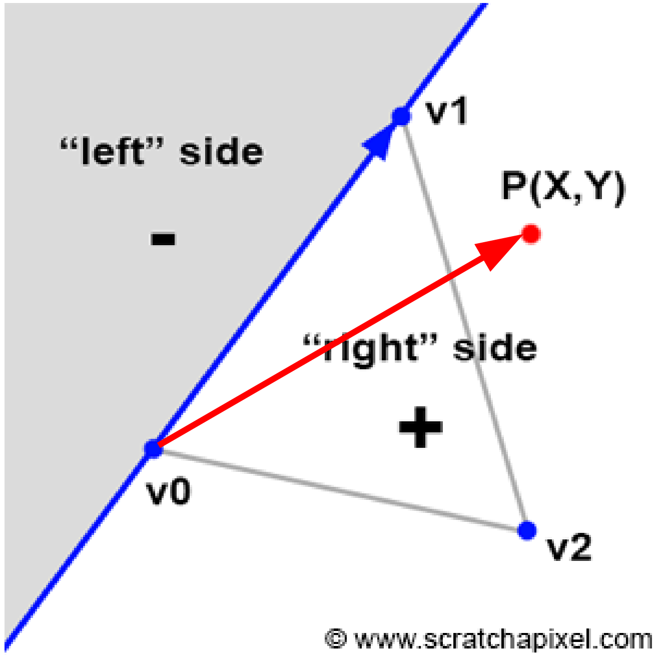
Inside-outside test: edge function

- **Triangle edges can be seen as lines splitting planes**
- **Edge function returns** (clockwise winding of triangle vertices):
  - **positive number if point P is on left side of the line**
  - **Negative number if point is on the right side of the line**
  - Zero if point is on 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)`

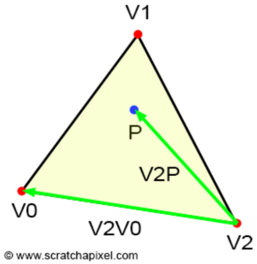


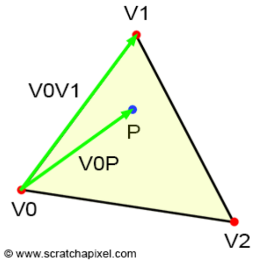
Reading: https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/rasterization-stage.html

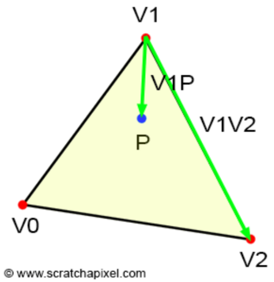
Inside-outside test: edge function test

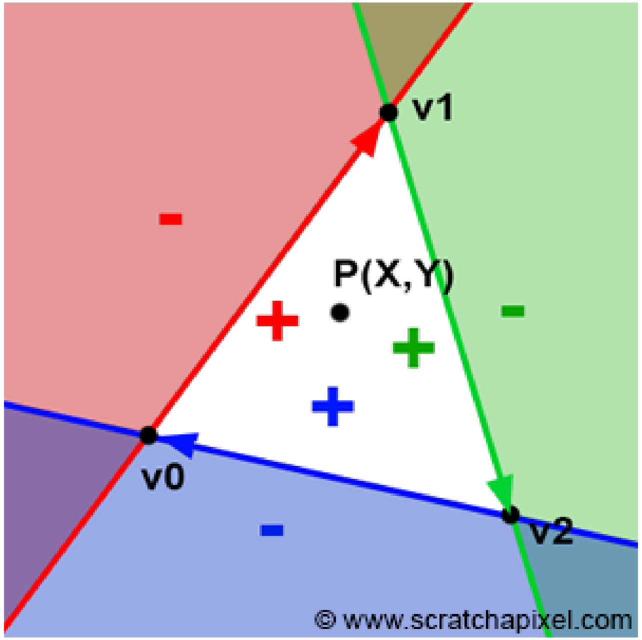
- Point must be tested for all edges
- If edge function returns positive value for all edges, then point is inside triangle
- Note: winding order of triangle matters
  - For counterclockwise winding, the sign of edge function is inverted

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









Barycentric interpolation and edge function

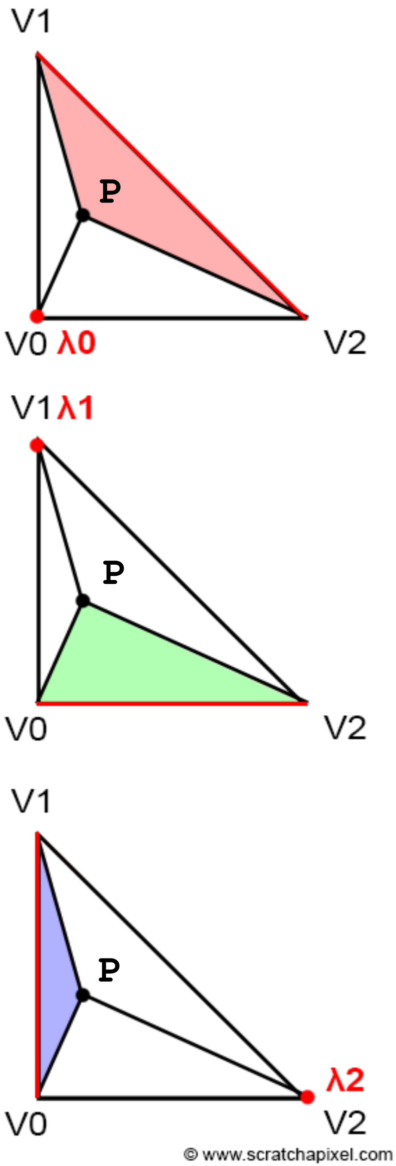
- Barycentric $(\lambda_1, \lambda_2, \lambda_3)$ coordinates can be used to define any point on the triangle.
  - Can be computed as the **ratio between area of sub-triangles and the whole triangle.**
  - **Area of sub-triangle is calculated using edge function.**

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

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

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

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



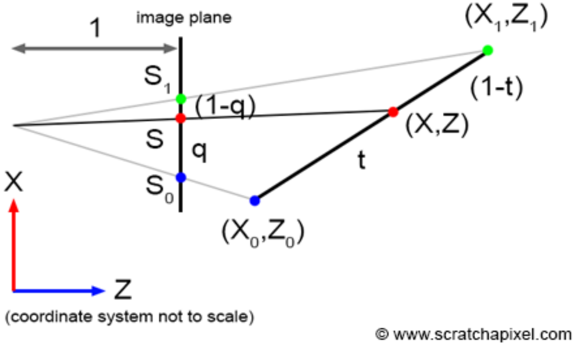
Depth and visibility

- **If pixel overlaps two or more triangles use z-buffer aka depth-buffer → visibility solving**
  - Compute z-coordinate or depth of the point on the triangle that the pixel overlaps
  - Compare computed depth with depth value stored at z-buffer at that pixel location
  - If current depth value is smaller than value from z-buffer, update frame-buffer with color of current triangle and z-buffer with depth of current point on triangle
- **Once all triangles have been processed, depth buffer contains distance to visible object from camera**
  - Depth is useful for visibility solving, shading and post-processing (e.g., depth of field, fog, shadow mapping, etc.)

Depth and barycentric coordinates

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

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



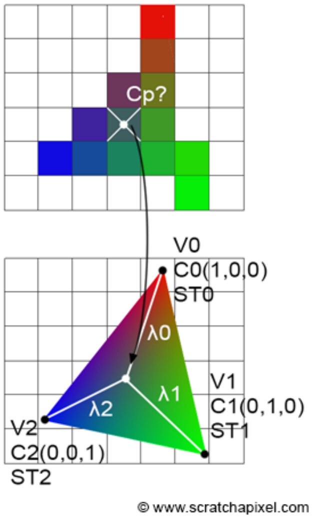
Perspective projection doesn't preserve distances.

Barycentric interpolation and vertex attributes

- **Rasterization stage ends once it is determined for all pixels if they overlap triangle** (inside-outside test)
- Triangle vertices can also have assigned **vertex attributes: colors, normals, texture coordinates, etc.**
- **Vertex attributes are used during shading** for computing color of pixels overlapping the triangle
  - Vertex attributes must be interpolated across surface of a triangle when it is rasterized
- **Barycentric coordinates** $(\lambda_1, \lambda_2, \lambda_3)$ **are used to interpolate vertex attributes** for any pixel overlapping the triangle
  - Example: interpolating color at point P where each vertex has color attribute: C_{v0}, C_{v1}, C_{v2}

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

$$C_{v0} = (1,0,0)$$
$$C_{v1} = (0,1,0)$$
$$C_{v2} = (0,01,)$$



Barycentric coordinates: perspective projection

- Due to perspective projection, it is required to preform **perspective correction** for vertex attribute interpolation
- Example: perspective correct color vertex attribute interpolation

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

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

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

Rasterization-based rendering on GPU

TODO

- Rasterization-based rendering technique is deeply integrated in GPU rendering. Raytracing-based rendering is purely implemented on CPU
- Rasterization-based rendering can also be completely implemented g purposes this is useful to understand all the aspects of it. There are some special use-cases which also benefit from CPU implementation of rasterization based-rendering*.
- Almost all professional software which uses rasterization-based rendering is using GPU hardware implementation which can be further programmed using graphical APIs such as OpenGL, Vulkan, DirectX, Metal, etc.

* Cases when objects that are rendered are smaller than pixel. Those are advanced topics and are related to point rendering (https://www.cg.tuwien.ac.at/research/publications/2022/SCHUETZ-2022-PCC/) or micropolygon rendering (https://docs.unrealengine.com/5.0/en-US/nanite-virtualized-geometry-in-unreal-engine/ or https://graphics.pixar.com/library/Reyes/)

Graphics rendering pipeline: conceptual overview

Using rasterization for rendering

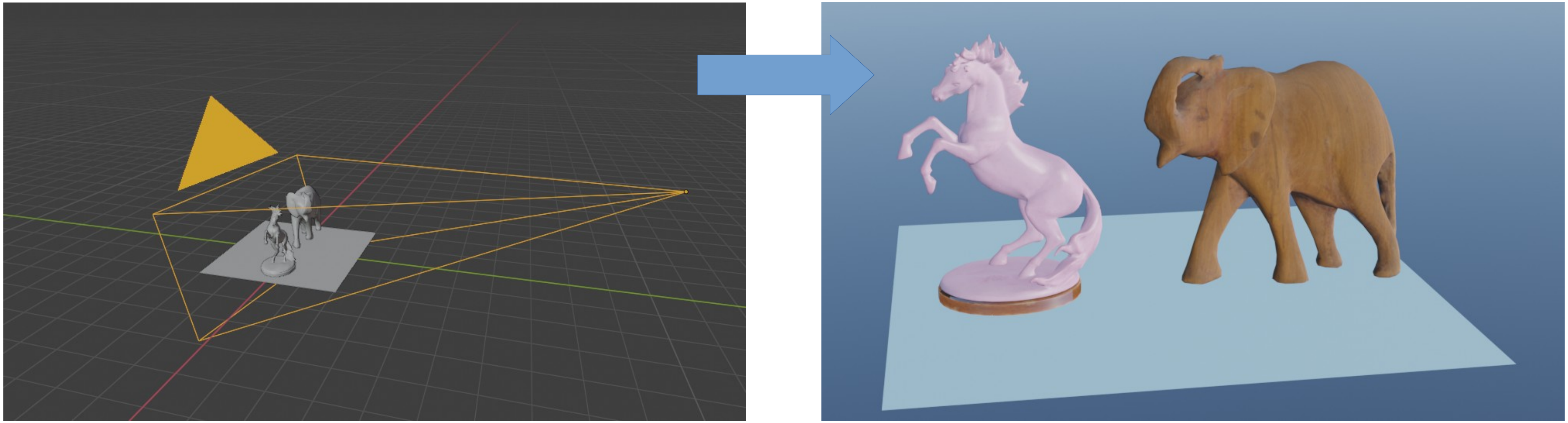
Motivation

- Graphics rendering pipeline describes rasterization based rendering on graphics processing unit (GPU)
- It is used for:
  - Real-time rendering applications
  - Interactive rendering a
  - Games, modeling, visu
  - VR, XR, etc.

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

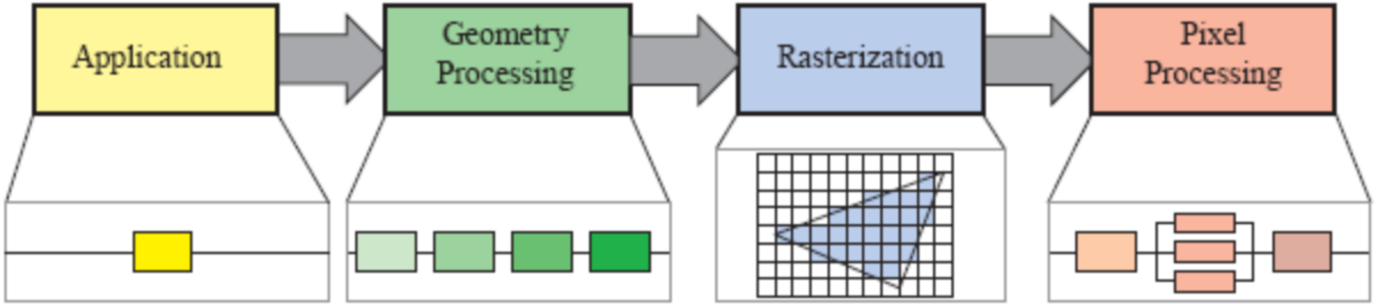


Graphics rendering pipeline

- **Graphics rendering pipeline is decomposed in smaller steps or stages**
  - Each stage performs **part of a larger task**
  - **Input of any given stage** depends on the **output of previous stage**
- **Sequence of stages forms rendering pipeline**
  - Pipeline stages, although working in parallel, are **stalled until slowest stage is finished**
  - Slowest stage is said to be **bottleneck**
  - Stages which are waiting are called **starved**

Graphics rendering pipeline: stages

- Graphics rendering pipeline can be **coarsely 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)**

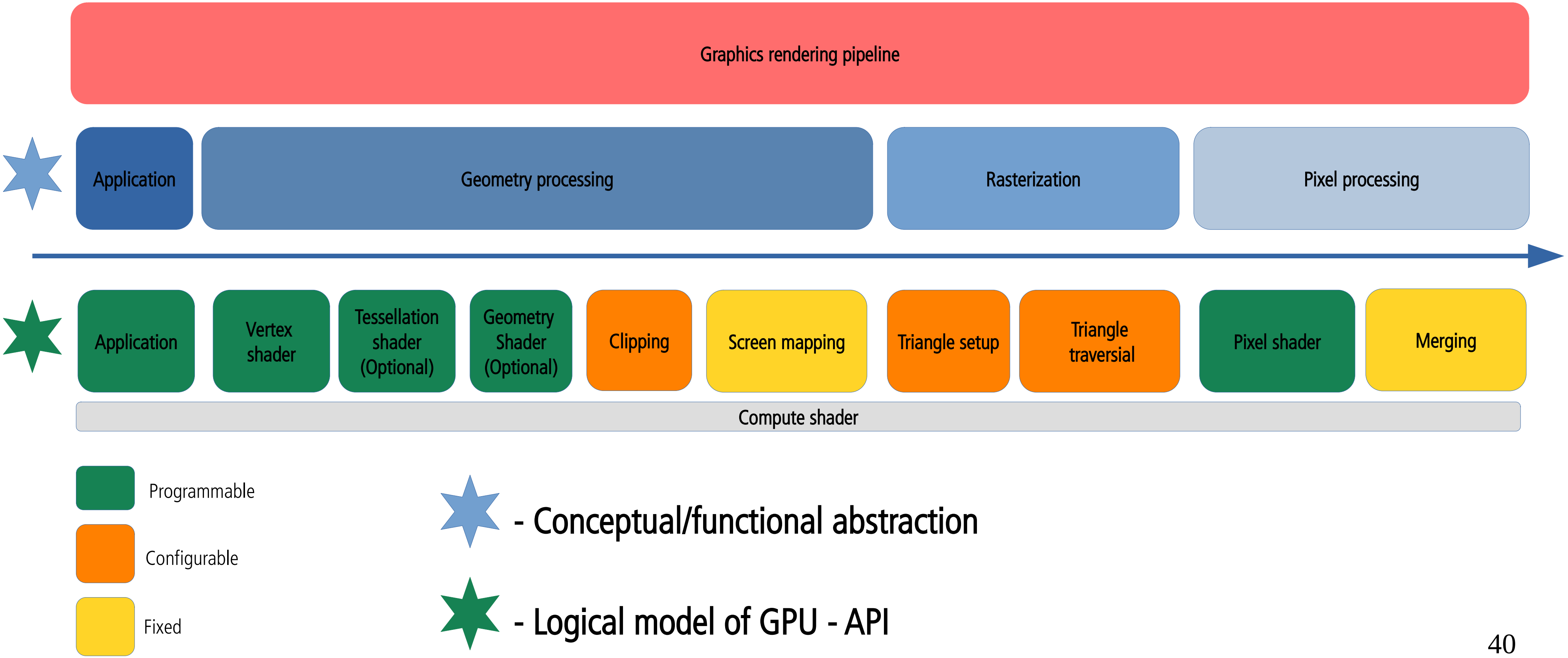
Trend is towards programmability and flexibility.

https://www.realtimerendering.com/

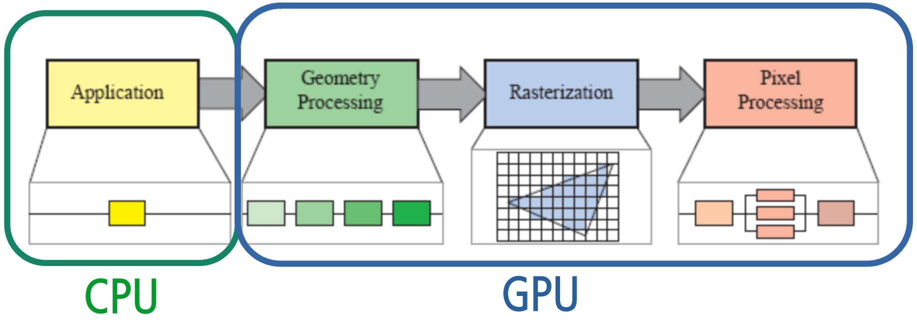
Pipeline abstraction

- **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 and exposed to the user as API. 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 vs logical (API) model

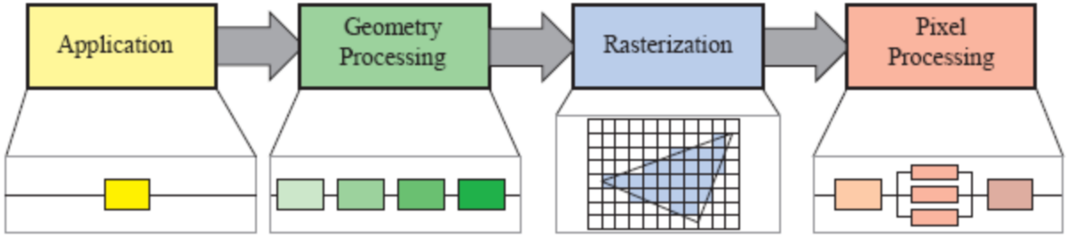


Pipeline: CPU and GPU

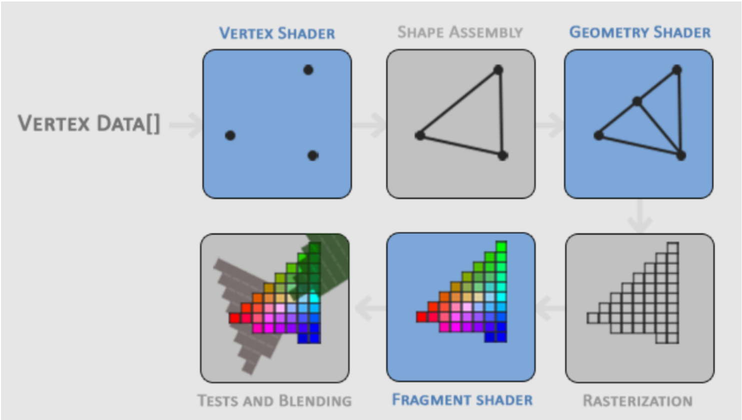


- **CPU implements application stage**.
  - CPUs are optimized for various data structures and large code bases, they can have multiple cores but in mostly serial fashion (SIMD processing is exception)
- **GPU implements** what is conceptually describes as **geometry processing, rasterization and pixel 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
  - 3D scene elements are defined here: objects (geometry, material), cameras, lights
  - Examples: 3D modeling tool application. User input is handled on application stage
- **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



https://learnopengl.com/Getting-started/Hello-Triangle

Example of pipeline in application

- Example: CAD program
- Application stage:
  - Enables user to select and move parts of the model. Selection and movement is done using mouse pointer. Thus, application stage must translate mouse movement into transformation matrix (e.g., translation or rotation) which is used in subsequent stage
  - Enables user to move camera in 3D scene to view the model from different angles. Thus, camera para … application.
  - For each frame (CAD programs are real-time when in modeling mode) application must provide inforr … primitives of the model.
- Geometry processing
  - Application provided parameters of the camera which includes projection matrix. Next, application, fo … orientation of the object.
  - Object is put in view space and optionally shading per vertices is preformed using provided light and
  - Projection is applied on the object tranforming it in unit cube space that represents what the eye sees … ecting this cube are clipped.
  - Finally, vertices are mapped into the window on the screen

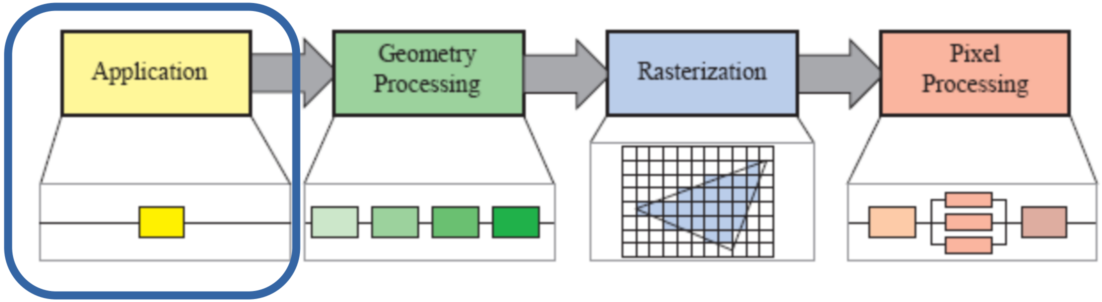
re-read

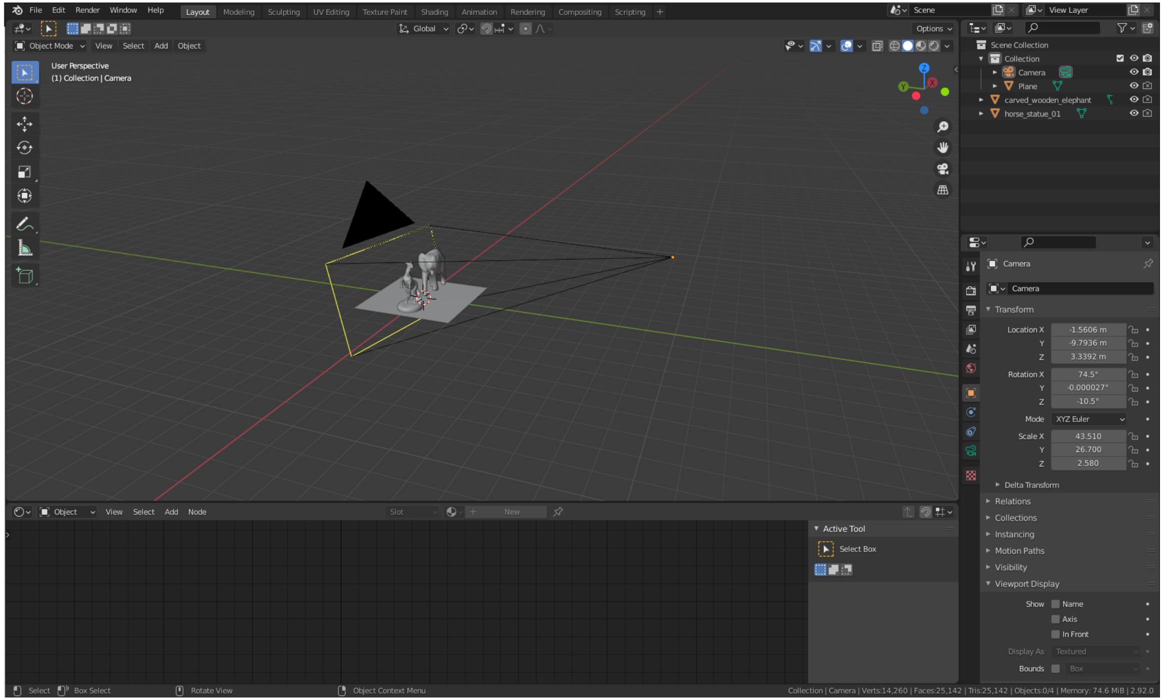
- Rasterization
  - All primitives coming from geometry stage are rasterized: all pixels (fragments) inside a primitive are found and sent for pixel processing.
- Pixel processing
  - Color of each pixel obtained from rasterization is computed using material information (colors, textures, shading equations) 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

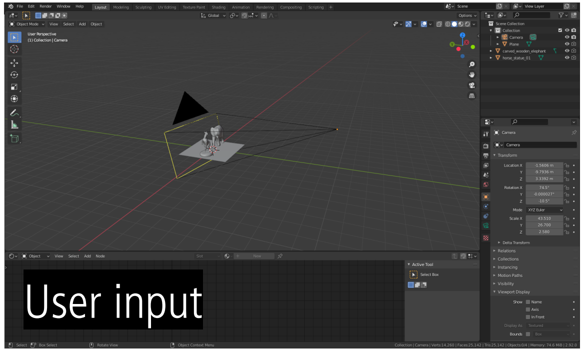
- Driven by application (e.g., 3D modeling application)
- Typically implemented on CPU (optionally on multiple threads)
- Developer has full control over what happens in this stage and how is what implemented
- Software-based implementation: not subdivided into stages

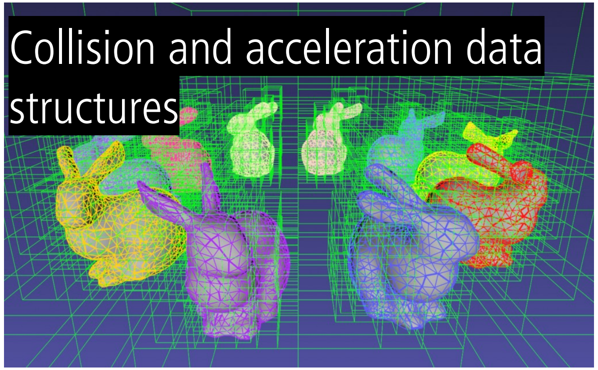




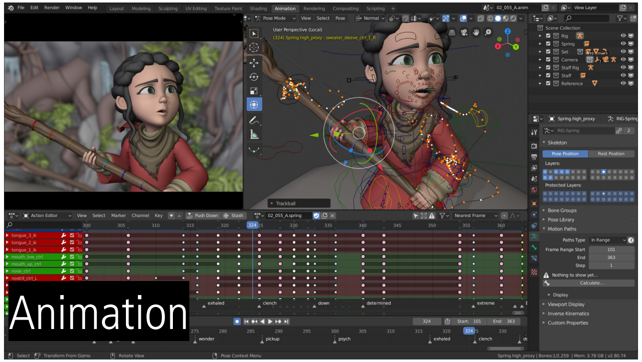
Application stage tasks

- 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





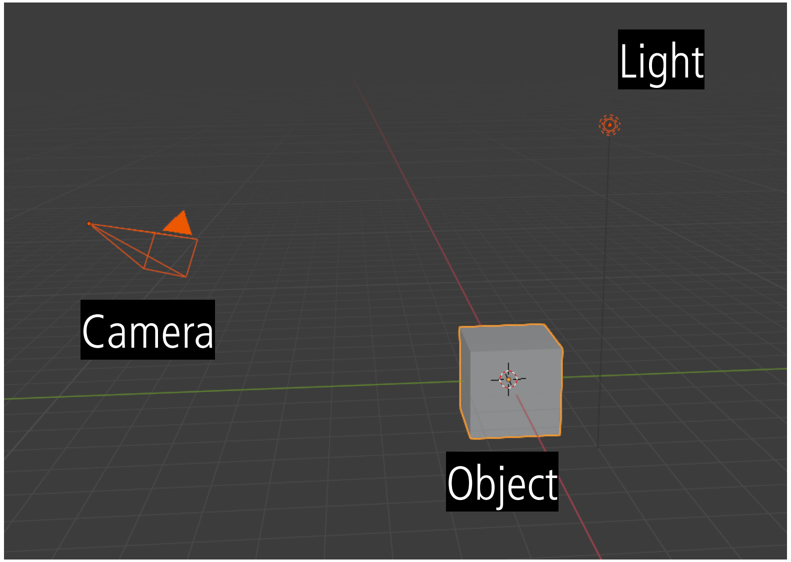
https://www.kitware.com/octree-collision-imstk/



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

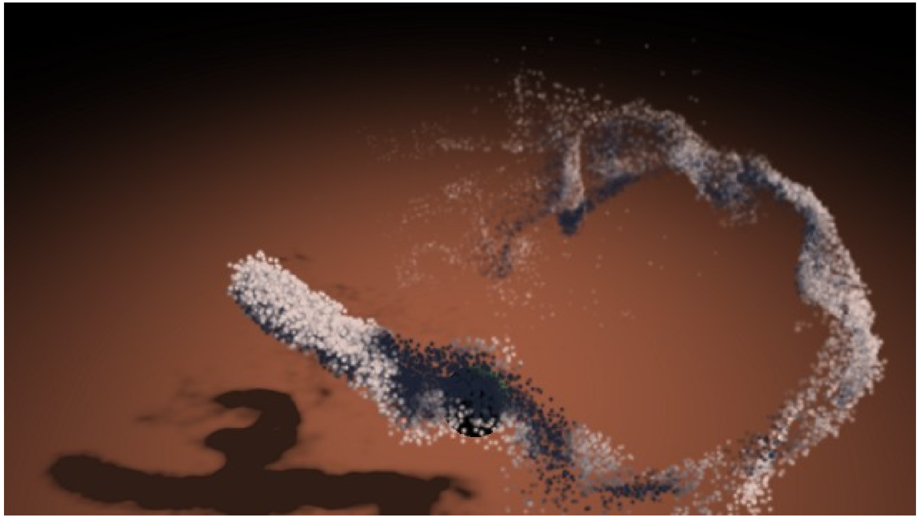
Application stage: scene definition

- 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)

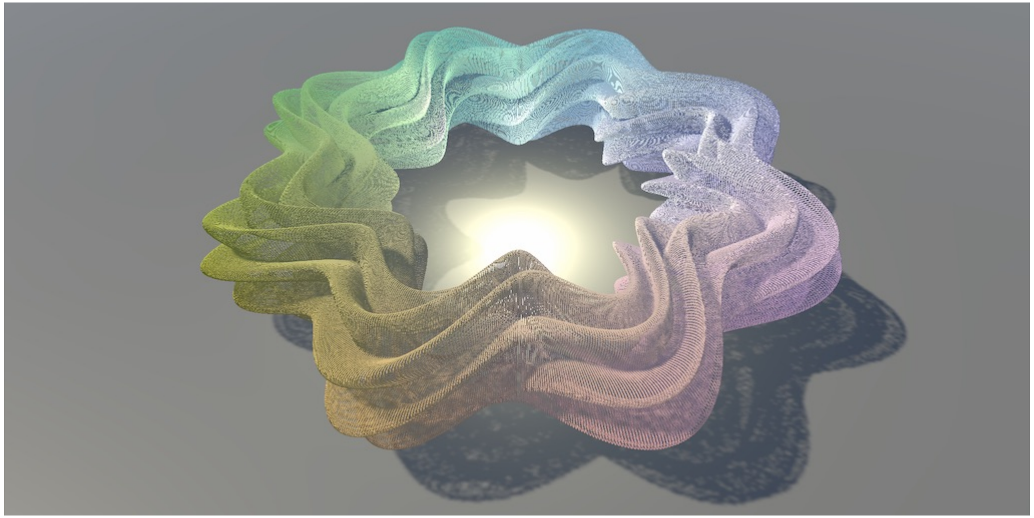


Application stage: compute shader

- 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

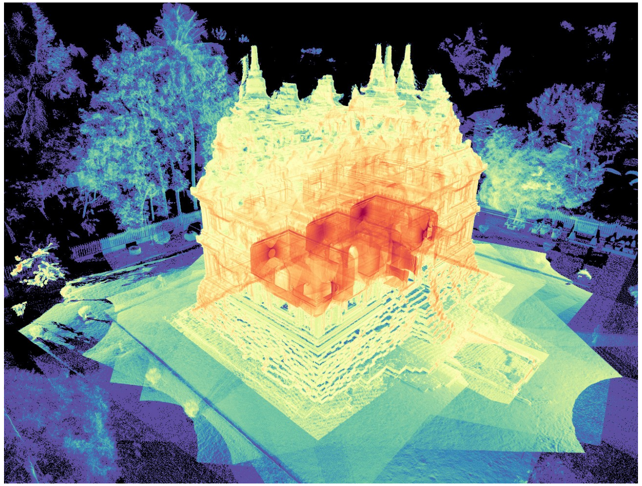


Compute shaders for point clouds animation:
https://catlikecoding.com/unity/tutorials/basics/compute-shaders/

Compute shader

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

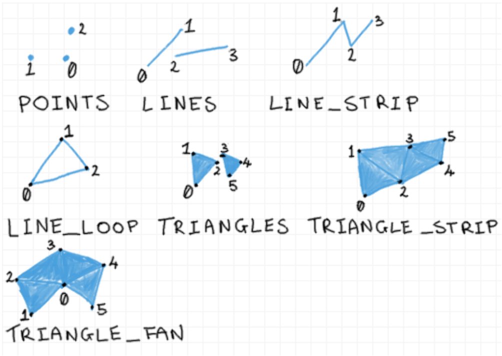
- **Compute shader is form of GPU computing** which is not necessary rendering
  - CUDA and OpenCL are used to control the GPU as massive parallel processor
- **It is closely tied to the graphics rendering but not locked into a specific location in the graphics pipeline**
  - It is used alongside vertex, 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

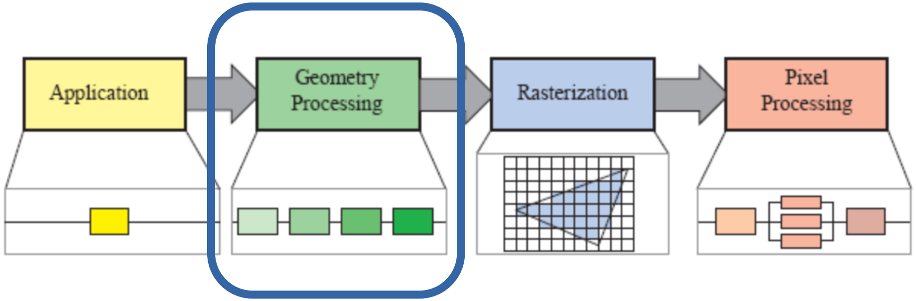


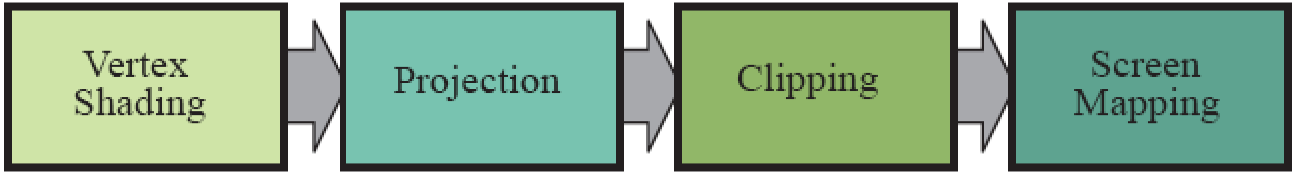
https://antongerdelan.net/opengl/vertexbuffers.html

2. Geometry processing stage

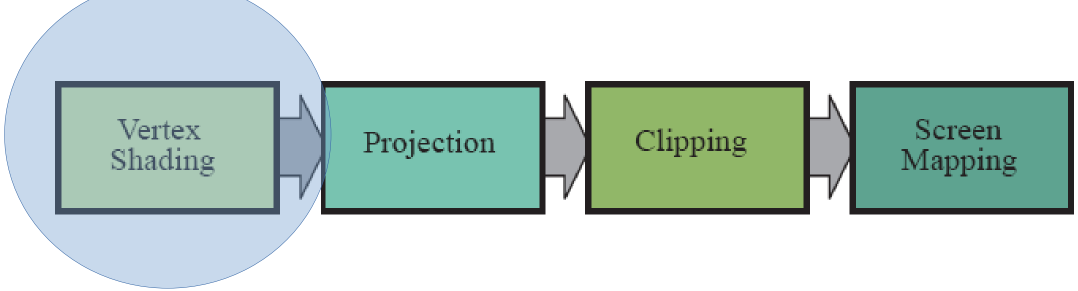
2. Geometry processing stage

- Responsible for most of the **per-triangle** and **per-vertex** geometry operations
  - Deals with **transformations**, **projections** and all other geometry handling
- Computes what is drawn, how and where
- Divided into following functional stages:
  - 2.1. **Vertex shading**
  - 2.2. **Projection**
  - 2.3. **Clipping**
  - 2.4. **Screen mapping**





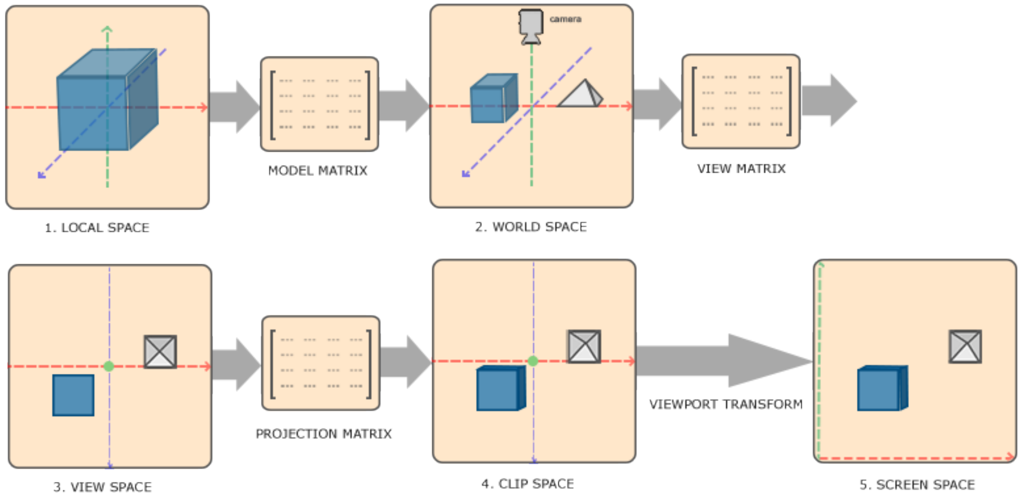
2.1 Vertex shading



- Two main tasks:
  - 2.1.1 **Compute 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.

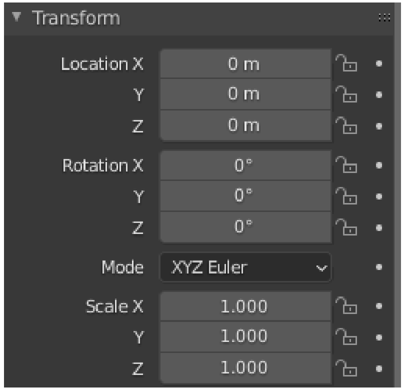
2.1.1 Vertex shading: compute vertex position

- **Vertex positions** of a model are minimal information that has to be passed from application stage to vertex shading stage.
- On the way to rasterization stage, **vertex shading performs transformation of a model in several different spaces** or coordinate systems:
  - **Model/local space**
  - **World space**
  - **View/camera space**

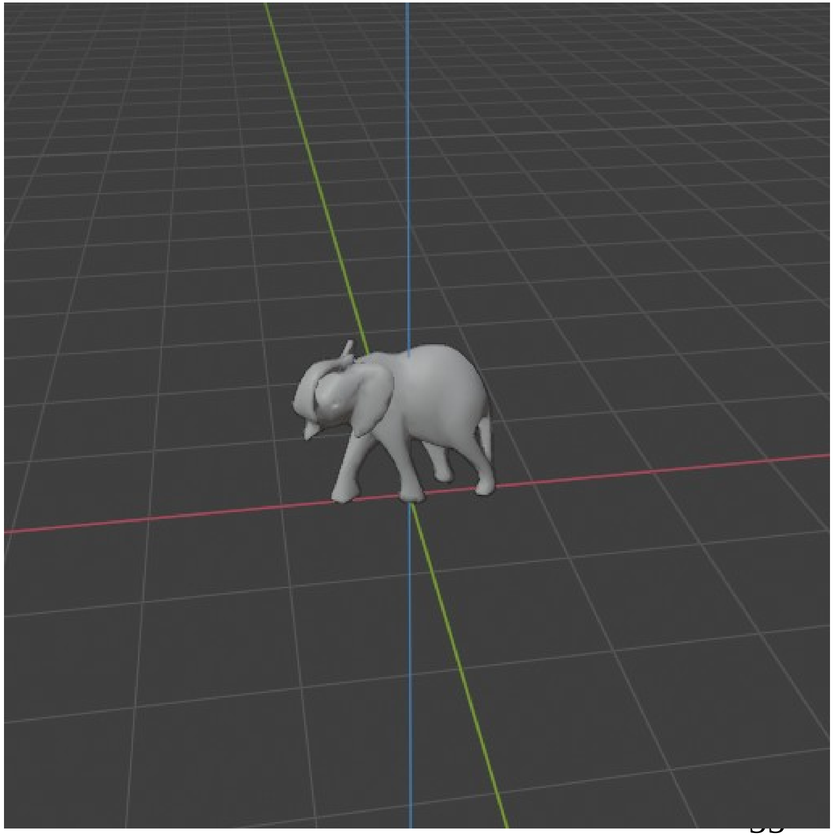


Model space

- Originally, model (vertex positions) resides is in its own **model space**.
  - Model has not been transformed at all
- Each model can be associated with **model transform**
  - Transformation matrix used for positioning and orienting
  - Used for model-to-world transformation
- **Model transform is applied on model's vertices and normals**
  - Coordinates of a object are also called **model coordinates**

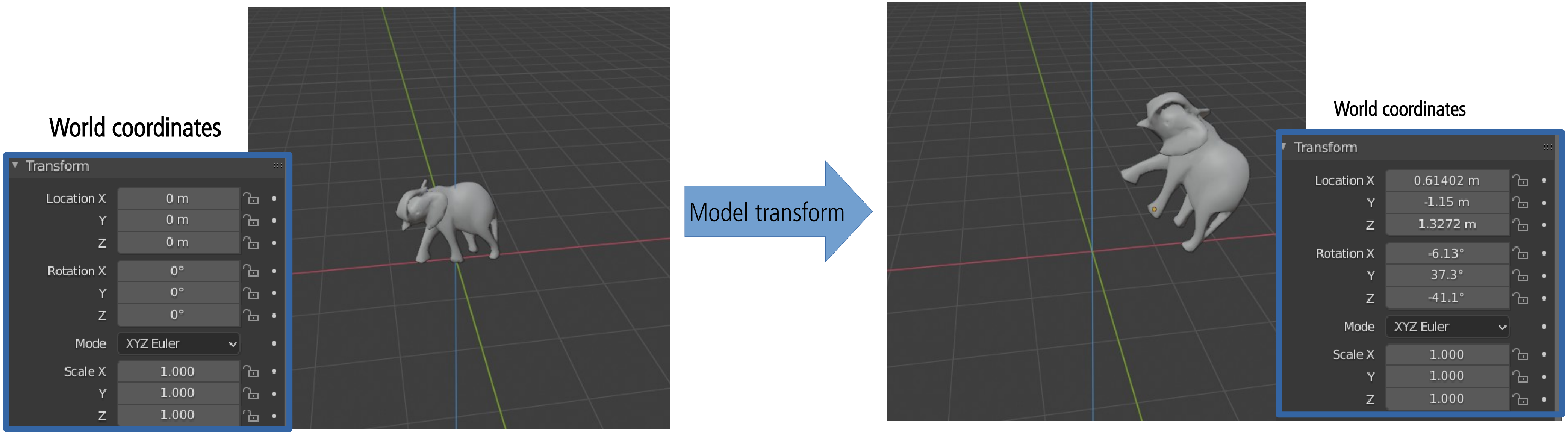


World coordinates



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**, sent to GPU and **performed by vertex shader**.



Model transformation matrix

$$\mathbf{T}(\Delta x, \Delta y, \Delta z) = \begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

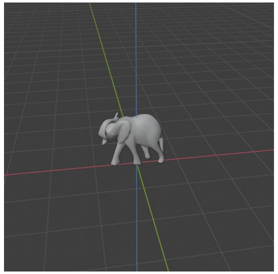
$$\mathbf{S}(x, y, z) = \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

$$\mathbf{R}_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

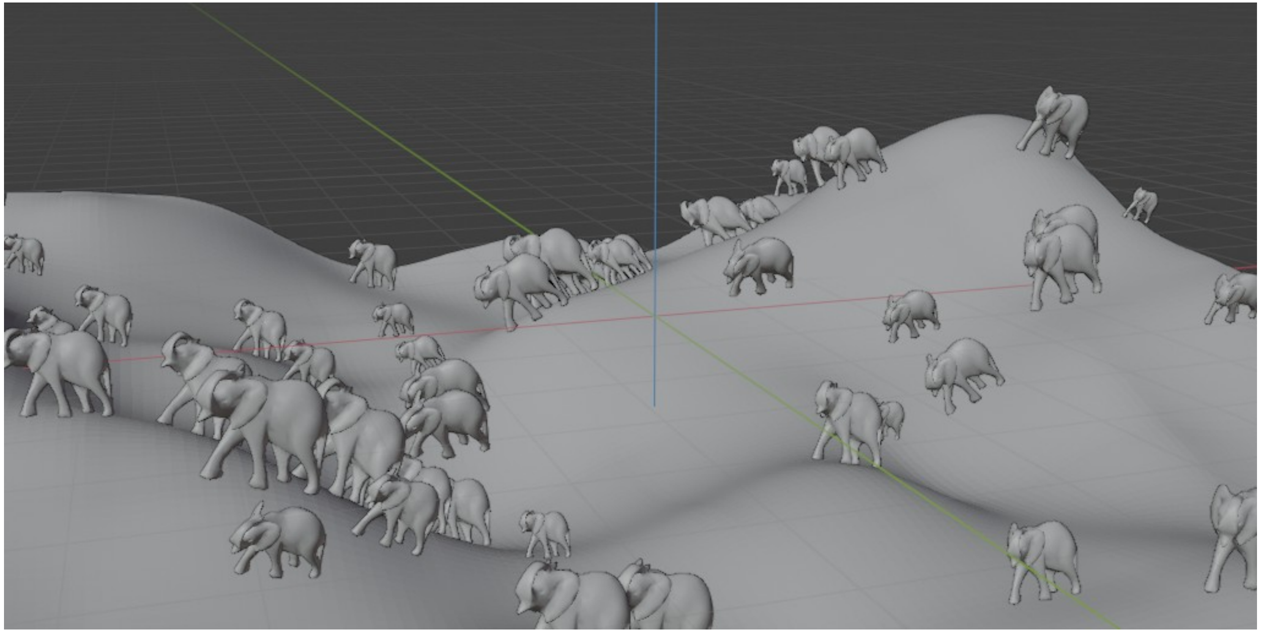
- Defines **model-to-world** of **world-to-model** transformation of mesh vertices
  - Model transform matrix applied on **model coordinates** gives **world space coordinates**
  - Inverse of transform matrix applied on **world coordinates** gives **model coordinates**
- Model transform is **4x4 matrix containing**:
  - **Translation (T)**
  - **Rotation (R)**
  - **Scaling (S)**
  - Other transforms: Euler, Quaternions, look-at transformation matrix, etc.
- Build transformation transformation matrices is supported by math libraries
  - Example: https://github.com/g-truc/glm

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.



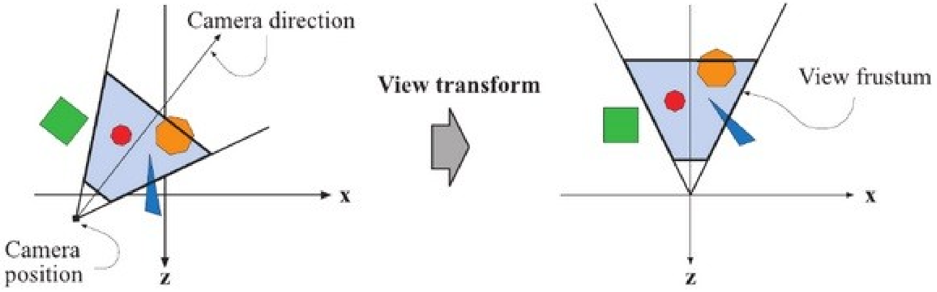
Instancing

- 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

- **Camera has a location and orientation in world space**
- **Only objects visible from camera are rendered**
  - Further pipeline stages (projection and clipping) require camera and all objects to be in **view (camera) space**
- **View transform** is applied to all objects and camera:
  - After transformation, camera is placed in world origin and aimed in negative z axis with y pointing up and x pointing right.
- View transform is defined on application stage and performed by vertex shader.

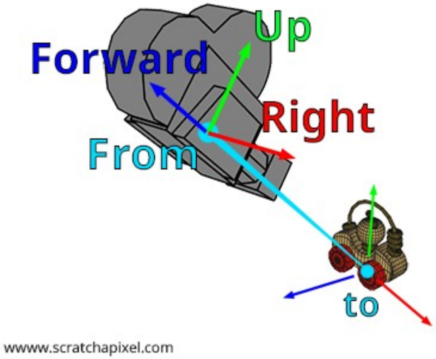


Negative z axis convention. Another convention is positive z axis convention. Actual position and direction after view transform are dependent on underlying API.

https://www.realtimerendering.com/

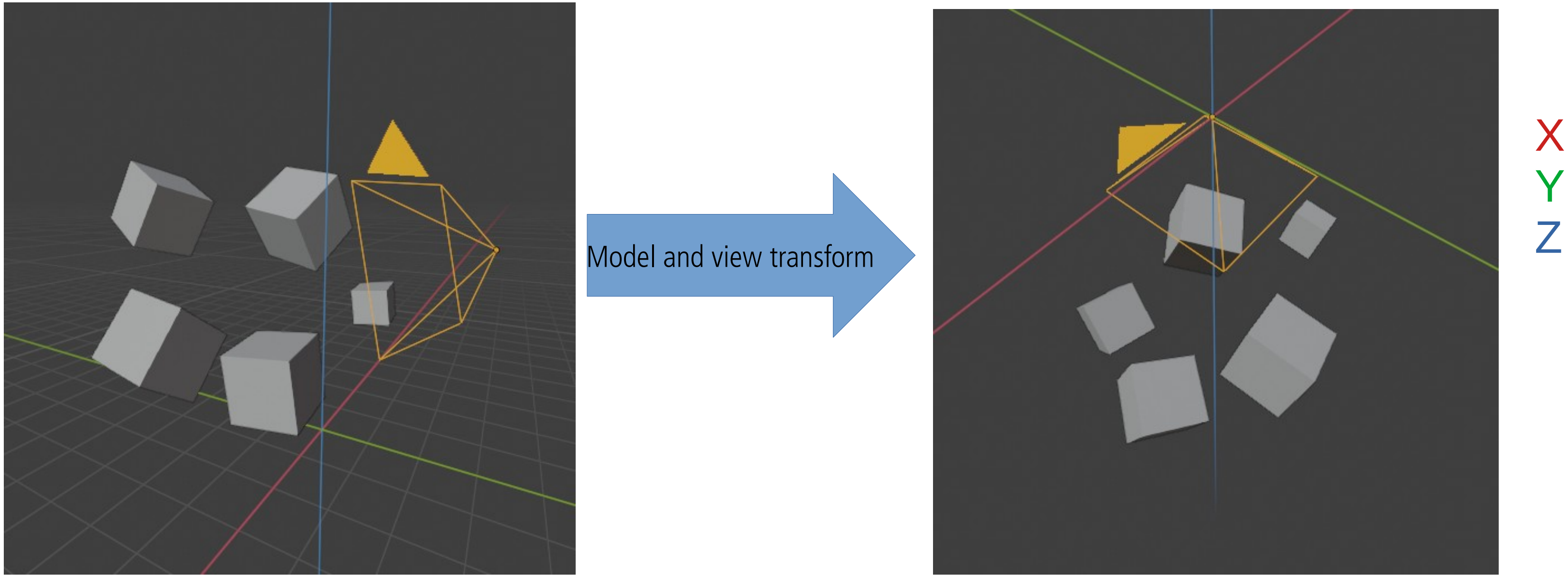
Camera in graphics pipeline

- Graphics rendering pipeline doesn't use camera as described in ray-tracing based rendering
- Camera position and orientation is used to create **view transform matrix** which is used transform all objects in 3D scene into camera space in which is needed for projection and sub-subsequent rendering stages
- **Various libraries exist for computing view matrix**
  - https://github.com/g-truc/glm/blob/master/manual.md#-52-glm-replacements-for-glu-functions
- Often, **view matrix** is constructed using **look-at notation**

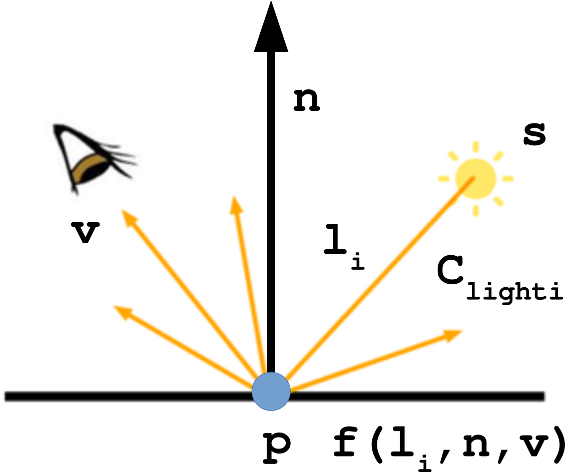


Camera (view, eye) space

- After **view** transform, the model is said to be in **camera (view or eye) space.**
- **Model and view transform can be implemented as one 4x4** matrix for efficient multiplication with vertex and normal vectors.
  - Note: programmer has full control over how the position and normals are computed.



2.1.2 Computing/setting-up vertex attributes



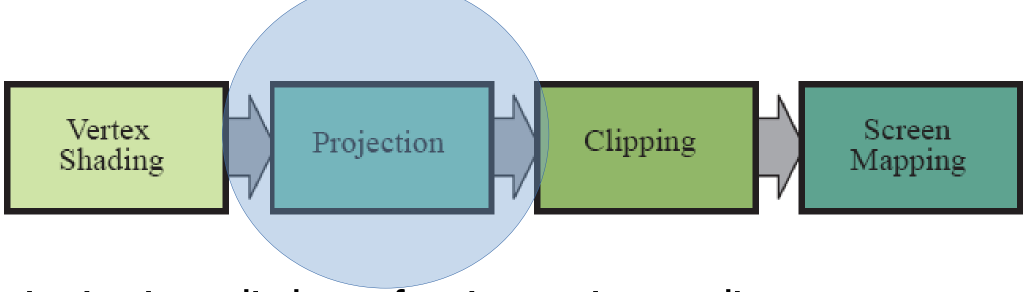
- **Vertex attributes: colors, texture coordinates, normals, etc.**
- **Shading** can also be performed on this stage → **vertex shading**
  - Computing color is done by **evaluating shading equation** using **vertex data** (position, normal, material) and **light information** defined on application 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

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

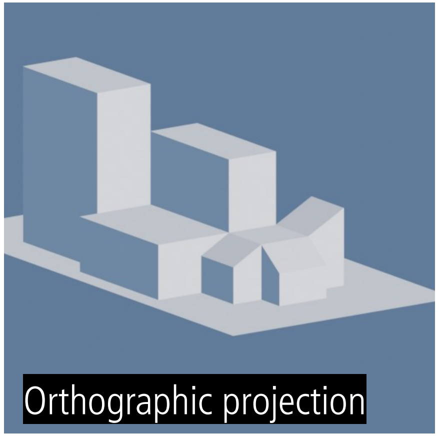
Light source color

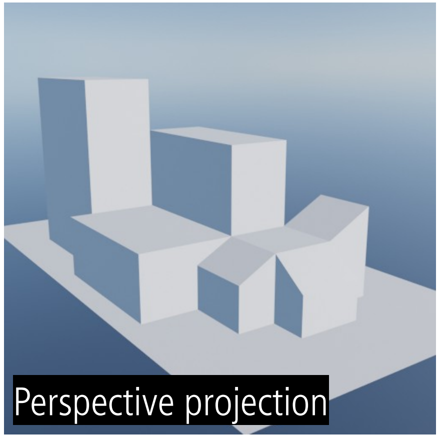
BRDF

2.2 Projection



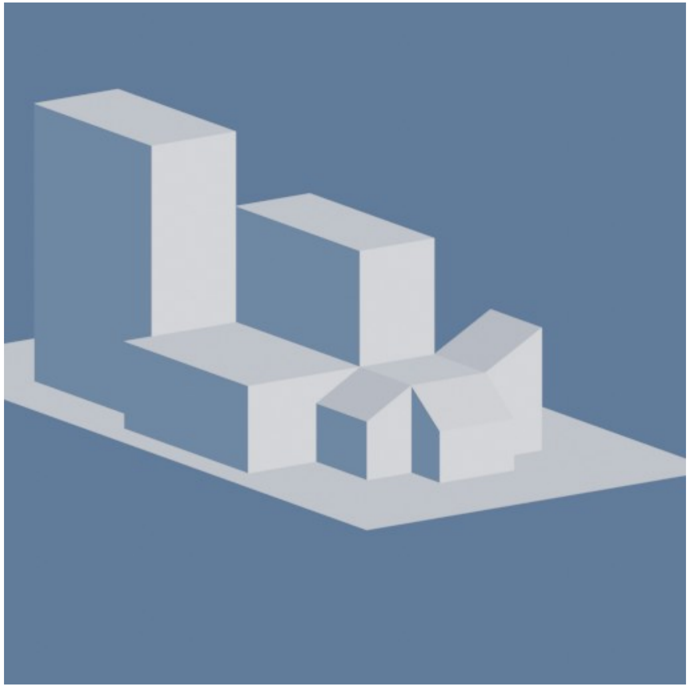
- **After models are transformed to camera space, a projection is applied transforming vertices to clip space**
- Two commonly used projection methods:
  - **Orthographic**
  - **Perspective**
- **Projection is represented as 4x4 matrix** and can be combined with other geometry transforms: model and view
  - **Projection matrix is defined in application stage and applied during vertex shading on object vertices**

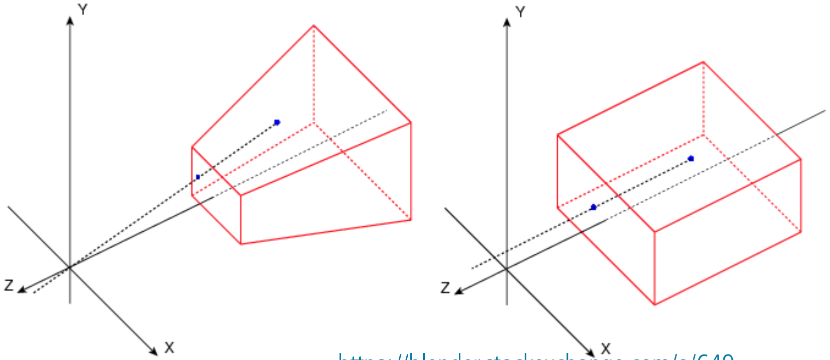

Orthographic projection


Perspective projection

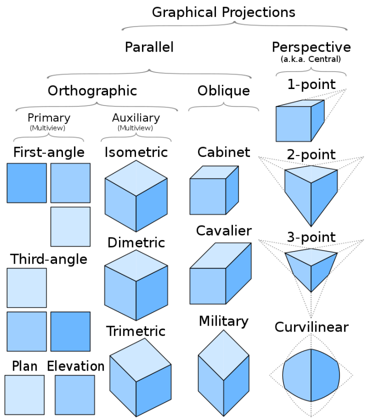
Orthographic projection

- Orthographic projection is just one type of parallel projection
- View volume of orthographic projection is rectangular box
- Characteristics:
  - parallel lines remain parallel
  - Objects maintain the same size regardless of distance from camera





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



Orthographic projection

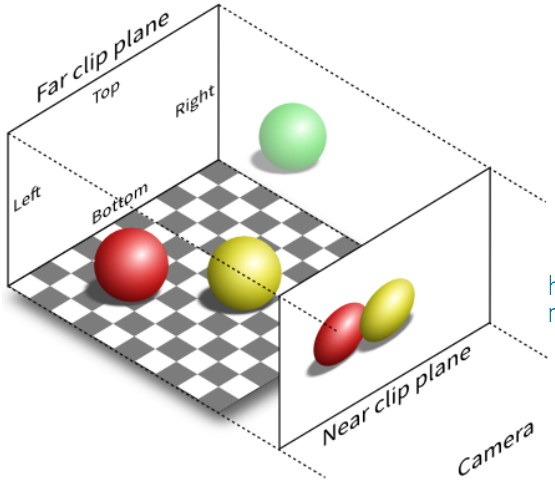
- **Simple orthographic projection** projects onto plane z = 0
  - View volume of orthographic projection is rectangular box
  - z coordinate is simply set to 0 → information of depth is lost!

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

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 matrix scales and translates axis aligned bounding box formed by these planes into axis-aligned cube centered around origin
  - In OpenGL*, minimum corner is (-1,-1,-1) and maximum (1,1,1) → **canonical view volume**

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



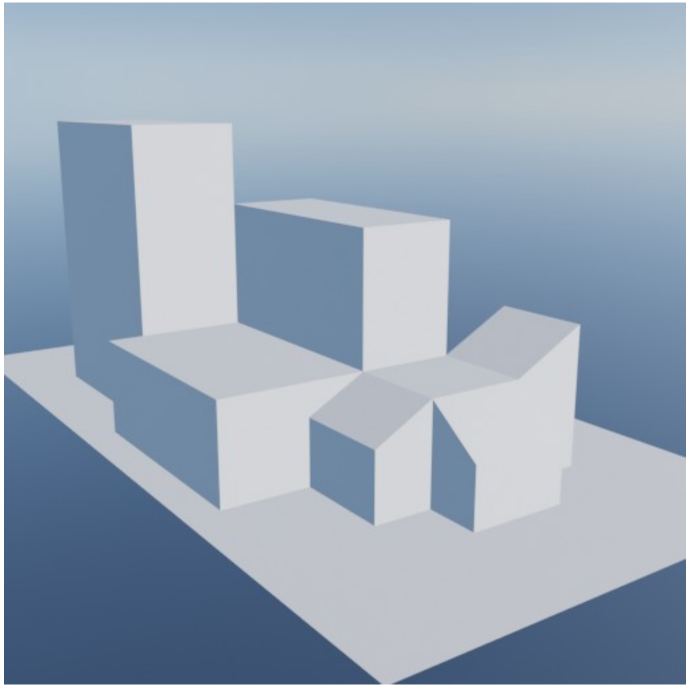
https://stackoverflow.com/q/36573283

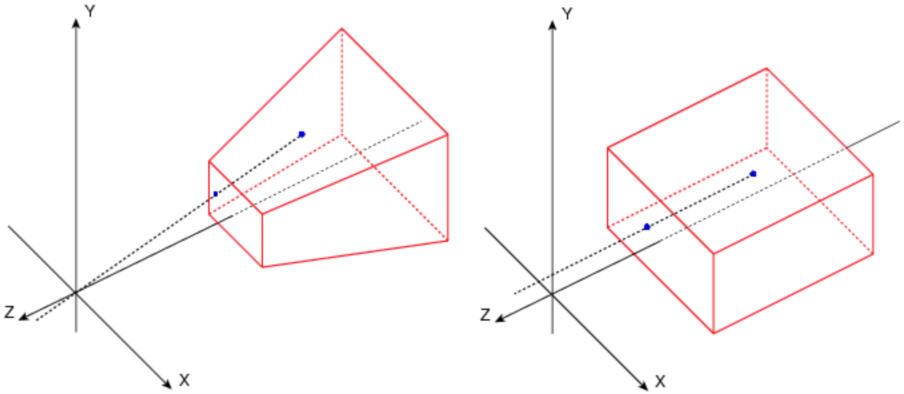
* DirectX has bounds (-1,-1,0) and (1,1,1)

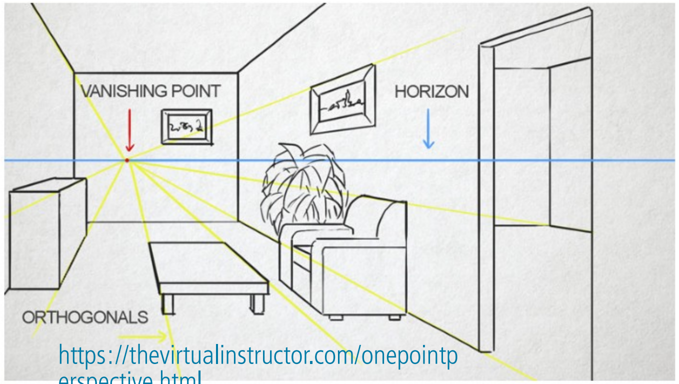
Derivation: https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/orthographic-projection-matrix.html

Perspective projection

- The view volume is called **fustum**
  - Truncated pyramid with rectangular base
- Characteristics:
  - The further the object, the smaller appears
  - Parallel lines converge at single point → mimics how humans perceive objects







https://thevirtualinstructor.com/onepointperspective.html

Perspective projection

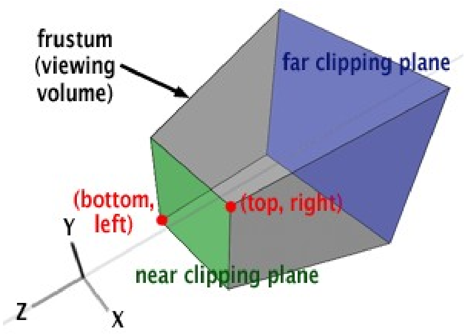
- **Simple perspective projection** is projecting points on a plane `z = 1` → information on depth is lost!
- After multiplication with perspective matrix, homogeneous coordinate w_c will be equal to vertex coordinate z. Other stays the same
- Perspective divide gives vertex on a plane

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

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

Perspective projection

- General perspective projection transforms viewing frustum to cube for which minimum corner is (-1,-1,-1) and maximum (1,1,1) → **canonical view volume**
  - General perspective projection matrix is defined with viewing frustum **near** (n), **far** (f), **left** (l), **right** (r), **top** (t) and **bottom** (b)
  - `OPENGL: glFrustum(float left, float right, float bottom, float top, float near, float far);`
- Construction of this matrix is supported by various libraries:
  - `OpenGL` Utility Library (GLU): `void gluPerspective(float fovy, float spect, float zNear, float zFar);` https://registry.khronos.org/OpenGL-Refpages/gl2.1/xhtml/gluPerspective.xml
  - GLM: https://github.com/g-truc/glm/blob/master/manual.md#-52-glm-replacements-for-glu-functions



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

OpenGL perspective projection matrix: column-major order

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

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

$$bottom = -top$$

$$right = top * aspect\ ratio$$

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

Camera's field of view (FOV) and image aspect ratio are used to compute left, right, bottom and top.

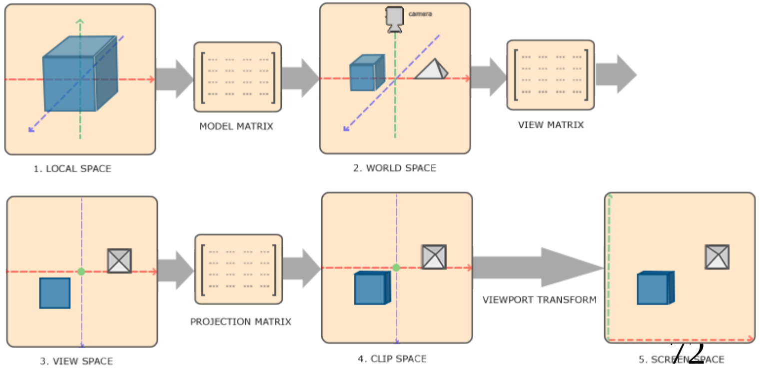
Derivation: https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/opengl-perspective-projection-matrix.html

Digression: camera

- Graphics rendering pipeline doesn’t have notion of camera as ray-tracing based rendering did
- To simulate camera projection (i.e., perspective projection) all objects in 3D scene are transformed with **projection matrix**

Projection

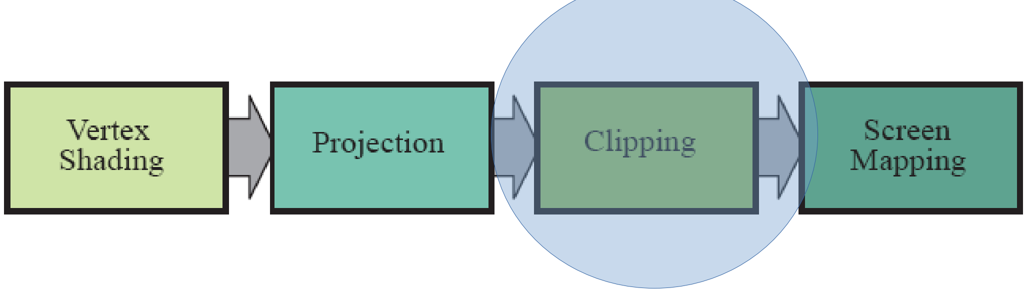
- After projection step vertex coordinates are in **clip space - clip coordinates**
  - These are homogeneous coordinates (x,y,z,w)
    - Perspective division with w to obtain (x', y', z') occurs later
  - Vertex shader must output this type for next, clipping stage
- z-coordinate is not stored in generated image but as a **depth-buffer/z-buffer**
  - Model is projected from three to two dimensions.



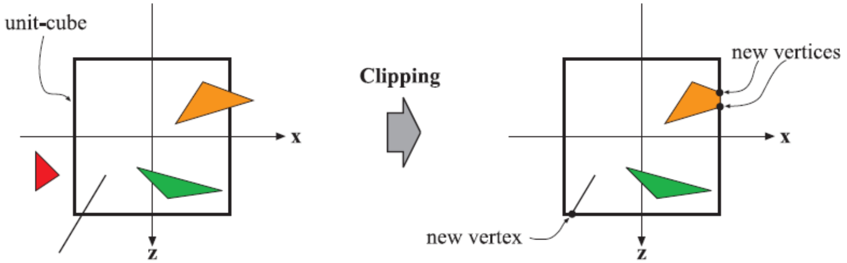
Optional vertex processing

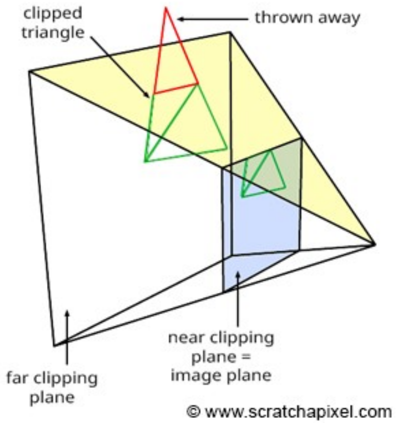
- Standard use of GPU’s pipeline is to send data through vertex shader, then rasterize the resulting triangles and process those in the pixel shader.
- Optional processing are:
  - **Tessellation** – curved surface can be generated with appropriate number of triangles. Sub-stages: hull shader, tessellator and domain shader
  - **Geometry shader** – takes in various primitives (e.g., triangles) and creates new vertices. Often used for particle generation – e.g., a set of vertices are given and square can be created for more detailed shading.
  - **Stream output** – instead of sending vertices down the pipeline, these can be outputted for further processing to CPU or GPU.
- Usage of those depends on GPU hardware (not all GPU supports those stages) and application.

2.3 Clipping



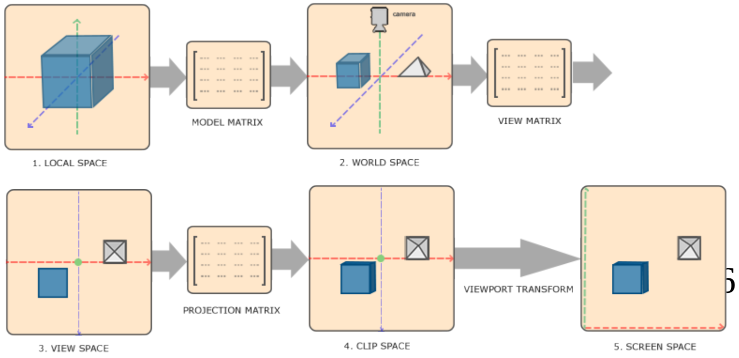
- Only primitives that are partially of fully in view volume need to be passed to the rasterization stage and pixel processing for drawing on screen.
- Primitive that is fully in view volume will be passed further without clipping
- Primitives that are partially in view volume require clipping.
  - After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





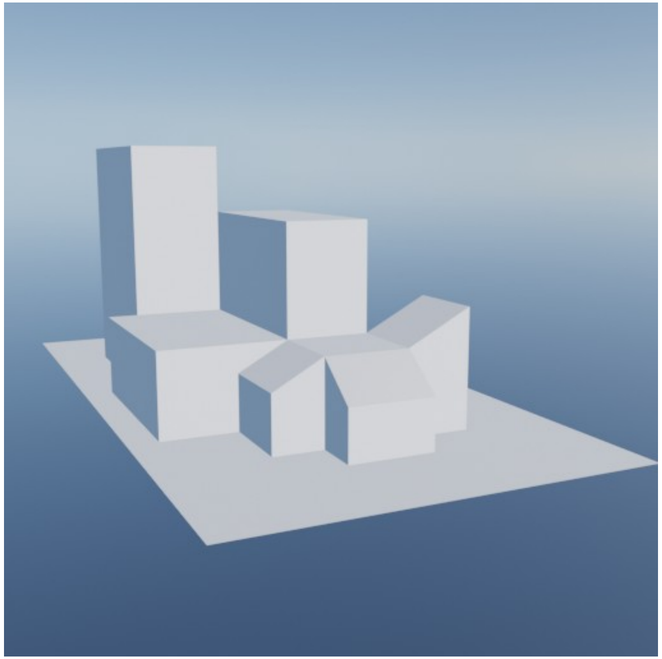
2.3 Clipping

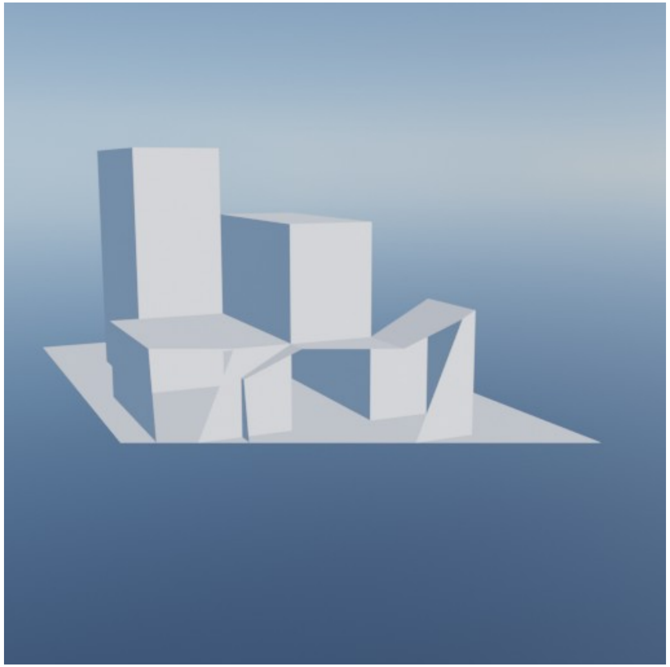
- Clipping uses clip coordinates (x,y,z,w) resulted from projection step.
  - Fourth coordinate (w) is used for correct interpolation over triangle and clipping if perspective projection is used.
- Finally perspective division is performed
  - Resulting triangle positions are in **normalized device coordinates (NDC)** – a **canonical view volume** that 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



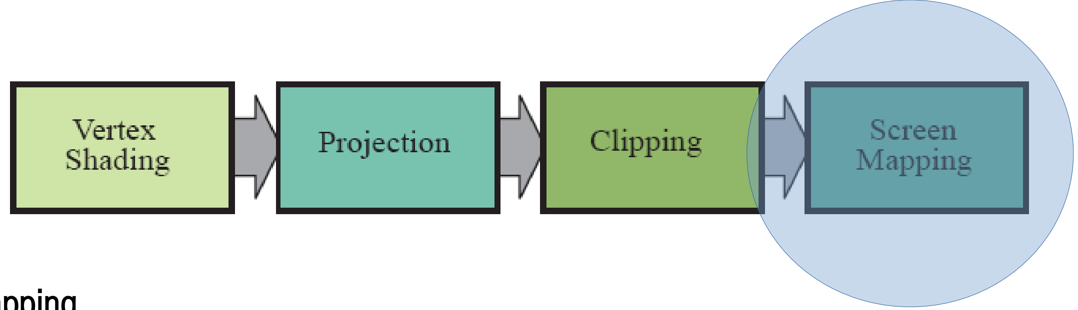
2.3 Clipping

- Additional clipping planes can be introduced by programmer to chop the visibility of objects: **sectioning**

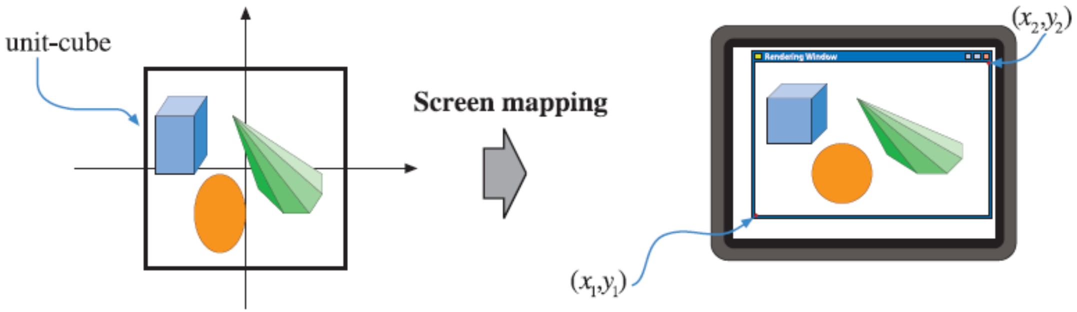




2.4 Screen mapping

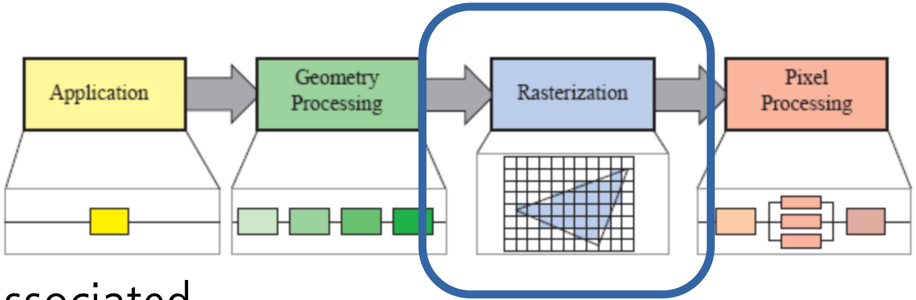


- Clipped primitives inside view volume are passed on **screen mapping**.
  - Coordinates entering this stage are still 3D → NDC.
- x and y coordinates are transformed to **screen coordinates**.
  - Screen mapping is translation followed by scaling to map (x,y) to screen with dimensions (x1,y1) and (x2, y2).
  - Pixel position: `d = floor(c), c = d + 0.5`
- Screen coordinates with z coordinates are called **window coordinates**.
  - Z coordinates are mapped to [-1,1] in OpenGL or [0,1] in DX.
- Window coordinates are passed to the rasterizer stage.

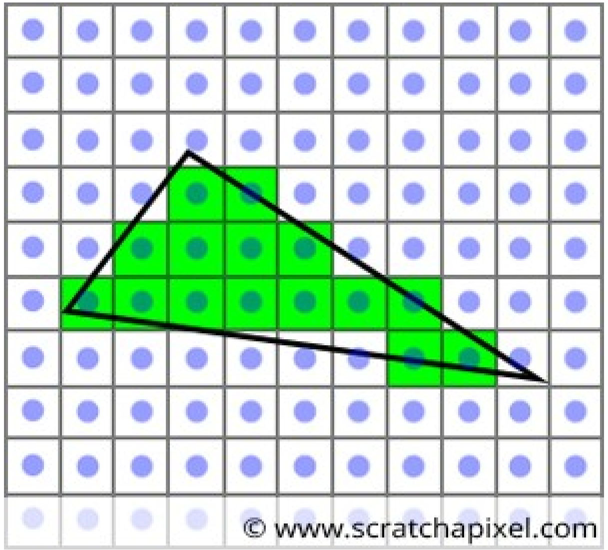


https://www.realtimerendering.com/

3. Rasterization stage



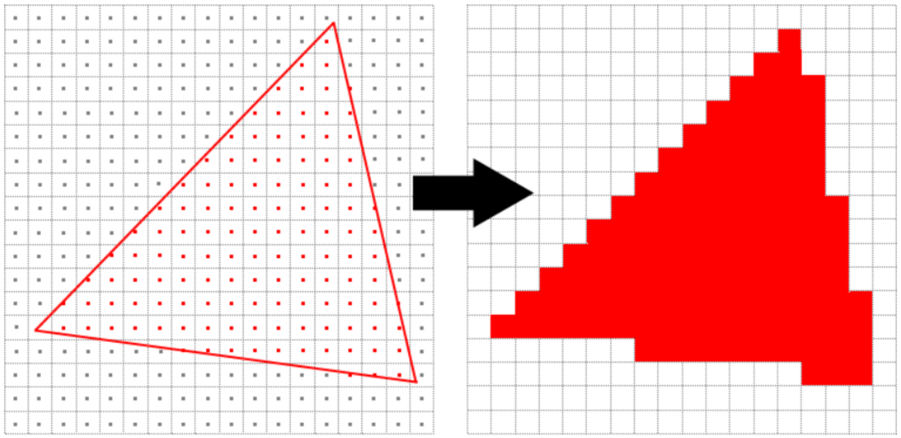
- Given transformed and projected vertices (with associated shading data) the goal is to find all pixels which are inside the primitive (e.g., triangle*) to be used in pixel processing stage → **rasterization**.
  - Typically takes input of 3 vertices forming a triangle, finds all pixels that are considered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



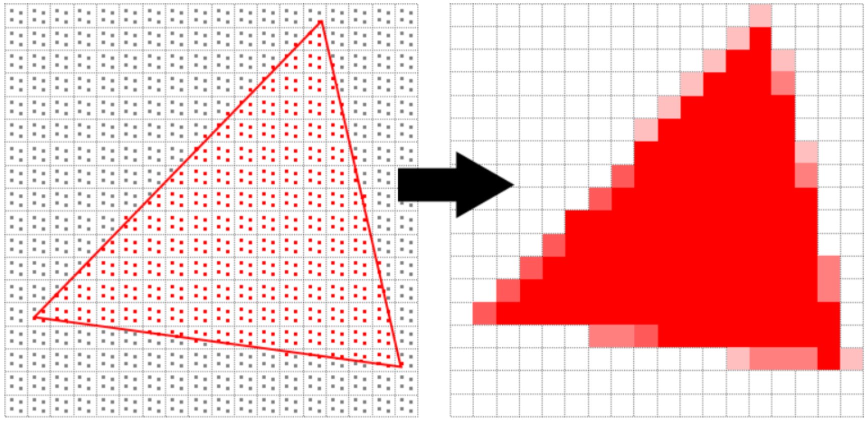
* Point and line primitives sent down the pipeline also create fragments for covered pixels.

3. Rasterization stage

- Test if triangle is overlapping a pixel, depends on GPU pipeline:
  - **Point sampling** - only center of pixel is used for testing.
    - If center of pixel is inside triangle → pixel is considered inside triangle
    - **Multiple samples per pixel** are desired to evade **aliasing** problems – **multi-sampling or anti-aliasing.**
  - Another approach is to define pixel overlaps triangle if at least part of pixel overlaps triangle.



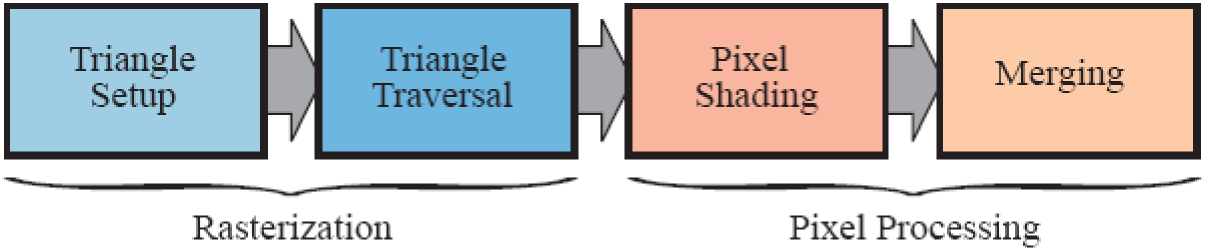
https://vulkan-tutorial.com/Multisampling



MSAAx4

3. Rasterization stage

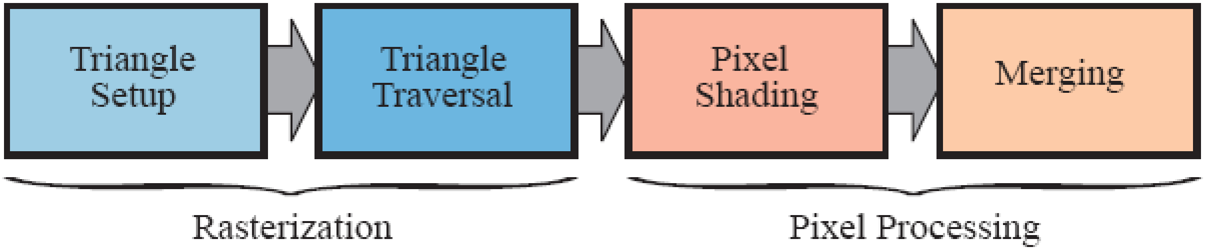
- Two functional sub-stages*:
  - 3.1.Triangle setup
  - 3.2 Triangle traversal
- Synchronization point between geometry processing and pixel processing



* Those can also process points and lines, but triangle is most often used primitive and thus the name.

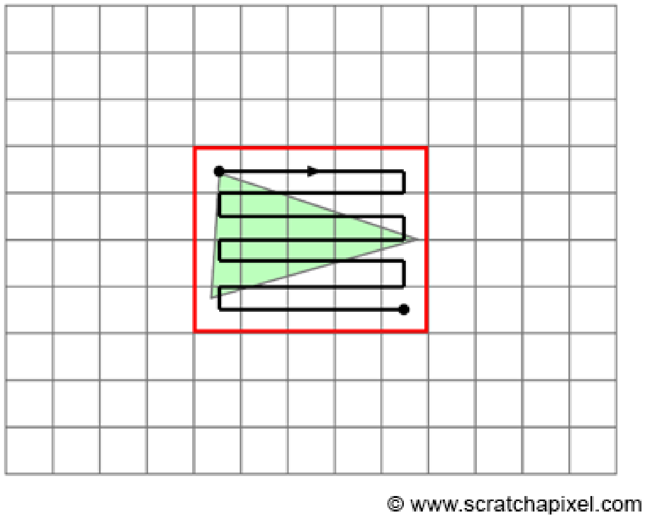
3.1 Rasterization: triangle setup

- Data for the triangle are are computed here (e.g., differentials, edge equations)
- This data is needed for:
  - Triangle traversal
  - 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.



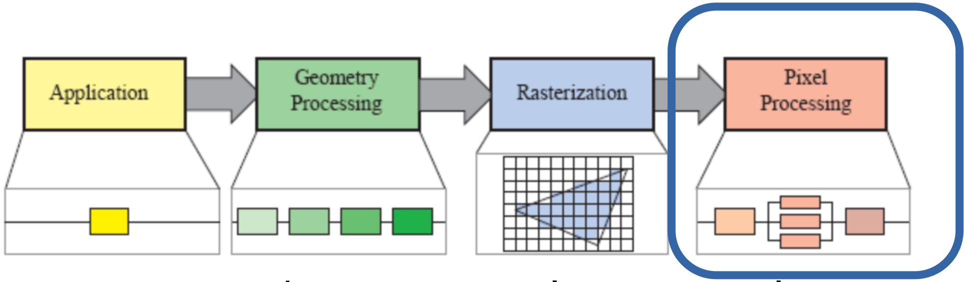
3.2 Rasterization: triangle traversal

- Each pixel sample is checked if covered by a triangle.
  - Finding which samples/centers for which pixels are inside a triangle is called **triangle traversal**
- **Fragment** is generated for part of the pixel that overlaps the triangle
- Fragment properties are generated by interpolating data among three triangle vertices, taking in account perspective – perspective-correct interpolation*.
  - Properties: fragment depth and any shading data from geometry processing stage.
- All pixels, that is fragments, inside primitive are sent to pixel processing stage.

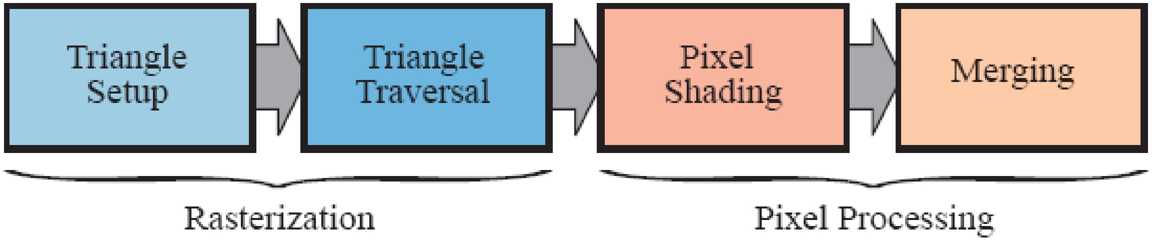


* Another type of interpolations are available; such as screen-space interpolation where perspective is not taken in account.

4. Pixel processing stage

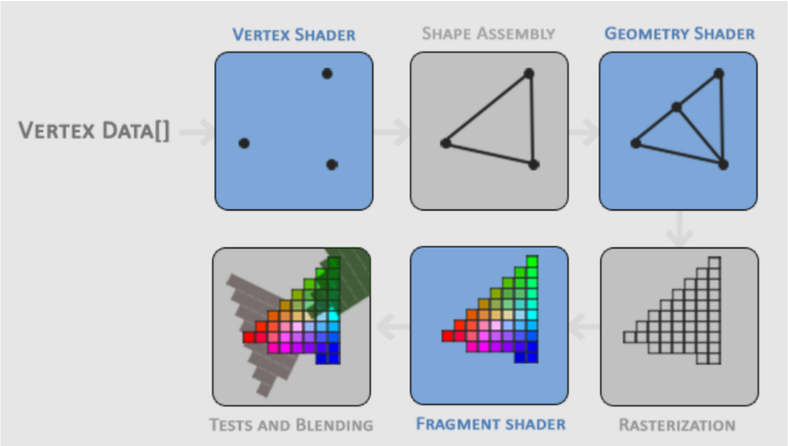


- Stage in which per-pixel or per-sample computations are done on pixels or samples inside a primitive.
  - A **shader program** is executed per-pixel/per-sample to determine its color and if color is visible (depth testing) as well as blending of pixel colors with newly computed with old color.
- Pixel processing stage can be divided into:
  - **4.1 Pixel shading**
  - **4.2 Pixel merging**



4.1 Pixel processing: pixel shading

- Any **per-pixel (fragment) shading** performs here
  - For computation, **interpolated shading data** from previous stage is used
- Pixel shading stage is performed by **programmable GPU cores**
  - Programmer supplies a **program for pixel (fragment) shader** that contain any desired computations.
  - Most commonly, **shading equations (scattering function)** and **texturing** (image or procedural) are defined here
- **Result of this stage is one or more colors for each pixel/fragment** that are passed further the pipeline



Pixel shading

- Here is shading defined
- Light, object and camera information is used here to compute color of pixel
- Rendering equation → d
  equation

todo

4.2 Pixel processing: pixel merging

- Information for each pixel (generated in pixel shader) is stored in **color buffer –** rectangular array of of colors `(r,g,b)`.
- This stage is also called **raster operations pipeline (ROP)** or **render output unit***.
  - It performs **raster operations** or **blend operations**
- This stage is not fully programmable, but highly configurable* for achieving various effects (e.g., transparency).

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

*Some APIs introduce more programmability, e.g. have support for raster order views – pixel shader ordering – which enables programmable blending capabilities (Real-Time Rendering Book).

Pixel merging: blending

- This stage is responsible to combine - **blend** - fragment color produced by pixel shading stage with the color currently stored in color buffer
  - Opaque surfaces blending is not needed: fragment color simply replace the previously stored color
  - Blending is important for transparent objects and compositing operations
- Color blending in particular can be set up to perform a large number of operations: combinations of multiplication, addition, subtractions, min, max, bitwise logic involving color and alpha values

Pixel merging: visibility

- This stage is responsible for resolving visibility
  - After rendering 3D scene, color buffer contains only color of primitives visible from camera point of view
  - Done using **z-buffer/depth-buffer**

Pixel merging: frame buffers

- This stage uses following buffers for computations:
  - **Z-buffer**
  - **Alpha channel** (part of color buffer)
  - **Stencil buffer**
- **Frame buffer** generally consists of all buffers on a system.

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 is same size and shape as color buffer, but for each pixel it stores **z-value to the closest primitive**.
- Z-value and color of pixel are updated with z-value and color of pixel being rendered
  - When a primitive is being rendered at a certain pixel, z-value at that pixel is being computed and compared to the z-buffer: if new z-value is smaller then one in z-buffer then that pixel is rendered closer to camera than previous pixel - which means updating color and z-buffer. Otherwise, color and z-buffer are not changed.

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

* Many algorithms require a specific order of execution. Often example is drawing transparent objects. In the standard pipeline, the fragment results are sorted in merger stage before being processed. DirectX API introduced rasterizer order views (ROV) to enforce order of exectution.

z-buffer

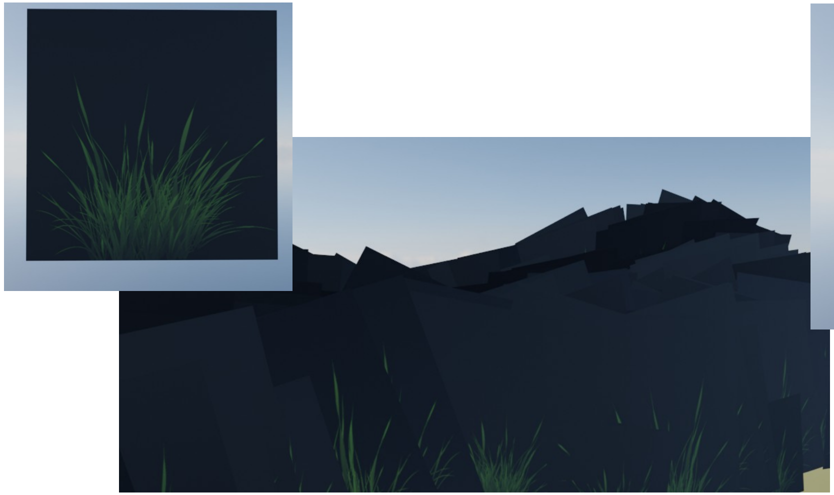
- Z-buffer algorithm is simple and of **O(n)** complexity, where n is number of primitives
- Z-buffer algorithm works for any primitive for which z-value can be computed, allows primitives to be rendered in any order and thus very much used – **order independent**.
- Z-Buffer stores only single depth value for each pixel – in the case of partially **transparent** objects; those must be rendered after all opaque primitives with end to front order* (or using order-independent algorithm), thus transparency is major weakness of z-buffer.

z-buffer

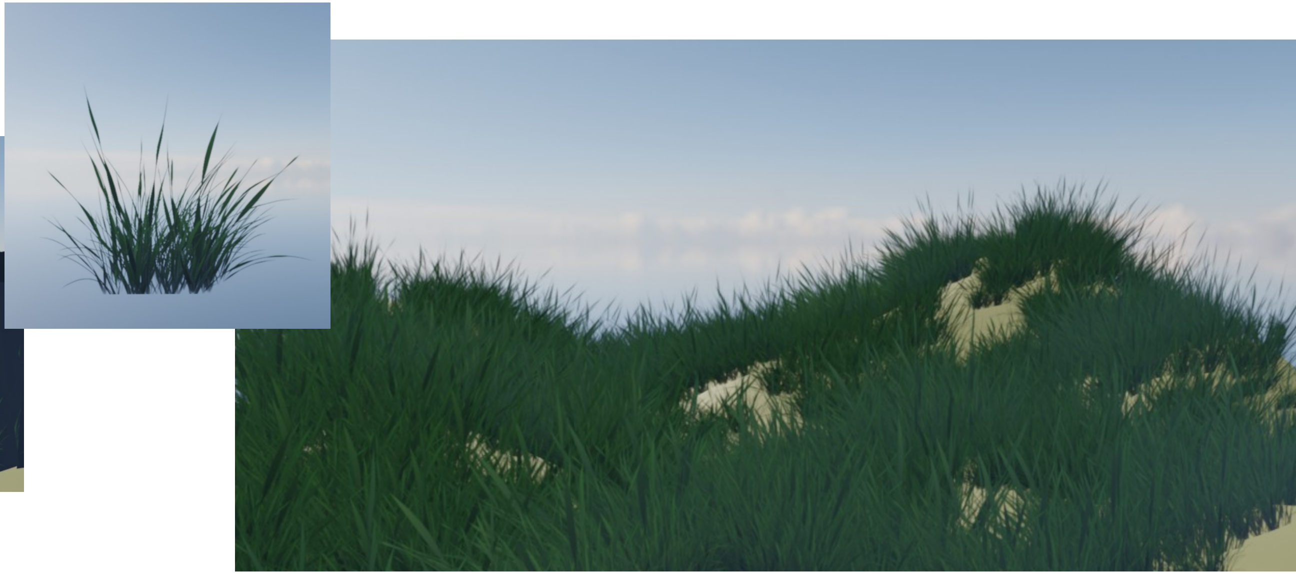
- Let’s recap the pipeline for now:
  - Fragments are generated in rasterization stage from 3 vertices forming triangle
  - Fragment is then run through pixel shader which computes color and blending information
  - Finally, in merging stage, this fragment is tested for visibility using z-buffer
- In third step, fragment can be discarded and all processing that was done was unnecessary.
- Therefore, 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 if hidden before pixel shader → **early-z** technique
  - Note that pixel shader can change z-depth of the fragment or discard whole fragment. In this case, early-z is not possible.

Alpha channel

- Frame buffer also contains information about **alpha value – alpha channel.**
- Alpha channel is associated with color buffer and stores opacity value for each pixel.
- Alpha value can be used for selective discarding of pixels using **alpha test** feature of pixel (fragment) shader.
  - Using this test, we can ensure that fully transparent pixels/fragments do not affect the z-buffer
- In modern APIs, alpha test can be done already in pixel shader

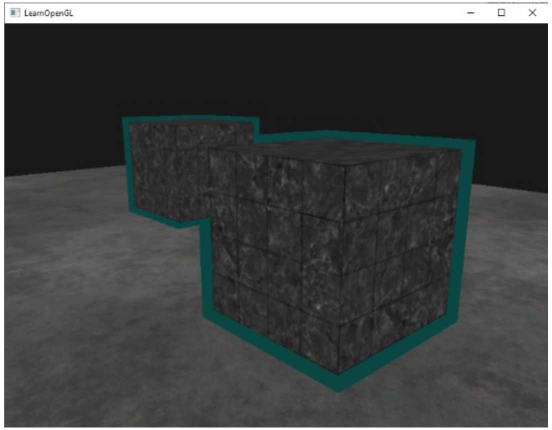


https://learnopengl.com/Advanced-OpenGL/Blending



Stencil buffer

- Stencil buffer is the type of so called **offscreen buffer*** – buffer that records locations of rendered primitives but not for directly showing on a screen rather used for pixel merging stage.
  - For 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 which are offers powerful tool for generating special effects by compositing rendering results.



https://learnopengl.com/Advanced-OpenGL/Stencil-testing

https://www.ronja-tutorials.com/post/022-stencil-buffers/

* This type of buffer is often used for advanced rendering (shading) techniques not only for stencil buffer.

Multiple render targets

- Instead of sending results of pixel shader's program to just color and z-buffer, multiple sets of values can be generated for each fragment and saved to different buffers – **render targets**.
- Multiple rendering targets functionality is powerful aid in performing rendering algorithms more efficiently.
  - Single rendering pass can generate a color image in one target, object identifiers in second and world space distances in third.
  - This inspired different type of rendering pipeline called **deferred shading –** visibility pass and shading are done in separate passes. First pass stores data about object's location and material at each pixel. Successive passes can efficiently apply illumination and other effects.
- Different buffers that are generated can be used for **compositing** purposes.

Digression: Note on pixel shader

- Pixel shader can write to a render target at only the fragment location handed to it – reading of current results from neighboring pixels is not possible – computations are performed only at given pixel.
- To solve this problem, output image can be created with all required data and accessed by a pixel shader in a later pass.
- Also, pixel shader is provided with the amounts by which any interpolated values change per pixel along x and y direction – gradient (derivative) information.
  - This is, for example, useful for texture filtering – where it is important to know how much of texture image covers a pixel.
- Graphics APIs (e.g., OpenGL, Vulkan and DirectX) are constantly evolving enabling more flexibility*.

* DX11 introduced a buffer type that allows write access to any location – unordered access view (UAV). OpenGL calls this buffer shader storage buffer object (SSBO).

Displaying pipeline output

- Primitives that have reached and passed rasterizer stage (remember that a lot of those are discarded) are visible from camera point of view
- Display device will show color buffer after all operations are done.
- As this takes some time (visible to human eye), a technique called **double buffering** is used.
  - Rendering of screen is performed in a **back buffer –** a off screen buffer.
  - Contents of back buffer are swapped with the contents of **front buffer –** buffer which is shown on display device.
  - Swapping process occurs* during **vertical trace** – a time when it is safe to do so.

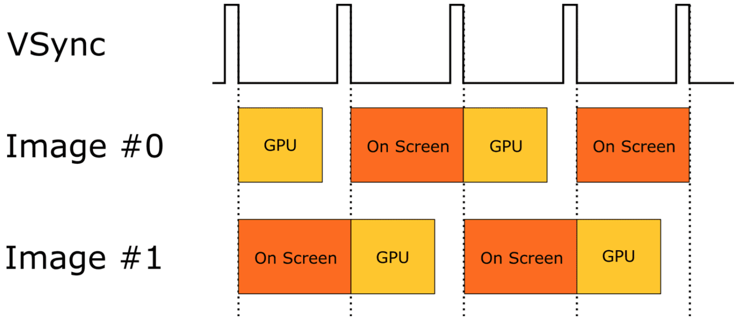


Figure 4: Correct double buffering behavior

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

More into topic

- 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

Summary questions

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

Repository

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