Rasterization-based rendering

Rasterization-based rendering: introduction

Introduction based on https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/overview-rasterization-algorithm.html

Rasterization: solution to visibility problem

- Rendering: visibility and shading.
- Raytracing-based rendering,
  - Camera rays visibility (which objects are visible from camera)
  - Secondary rays visibility (which surfaces are visible to each other during light transport in shading phase).
- Rasterization is another method for computing visibility.
  - Extremely efficient for finding objects that are visible from camera
  - Not very good when it comes to visibility needed for light transport.
  - Rasterization and ray-tracing should produce the same images until the point the shading is applied
- Calculation of visibility using rasterizer as it was in raytracing-based rendering, relies on geometrical techniques

Reminder: ray-tracing

- Reminder: ray-tracing-based rendering starts from image plane, generates rays which are traced into scene and tested for intersections. This method is call **image centric** and has two loops:
  - Outer loop that iterates over all pixels (and generates ray)
  - Inner loop that iterates over all objects in 3D scene (and tests for intersection)
- Now it is very important to do a quick recap on object shape representation. We discussed that ray-tracing based rendering can work with any shape representation as long as intersection of ray and the shape in question is defined. We also concluded (as it is done by almost all professional software) that user should be provided with shape representations which are easy to work with while renderer should be provided with the shape information which is tractable for rendering process.
  - Converting geometry to trangulated mesh makes process of rendering much simpler due to simplicity of triangle primitive.
  - Therefore, the shape representation which is almost always used for rendering purposes is **triangulated mesh**. Thus our rendering primitive is triangle and we will assume we always have triangulated mesh when we discuss rasterization-based rendering.

Rasterization

- To solve the **visibility problem**, rasterization takes the opposite approach to ray-tracing – it starts from objects (which are triangulated mesh) in 3D scene.
- First, it projects triangles (which compose the object) onto image plane (e.g., perspective projection). This is done by multiplying triangle vertices with **projection matrix**.
  - This will be done using perspective divide and remapping resulting coordinates to raster space.
- Secondly, a method is employed to compute which pixels of image plane are covered by the projected triangles. This is done by looping all pixels in the image and test if they lie within 2D triangles.
- The resulting pixels from rasterization are used for shading (which is separate process from rasterization)

Rasterization*: algorithm

```
for each triangle** in the scene:
    project vertices using perspective projection;
        for each pixel in the image:
            compute if pixel is inside the projected triangle;
```

- This algorihtm is object centric – it starts from object geometry and then uses image pixels.
- Represented algorithm is only the simplest form. In actual implementations, various optimizations are performed.
  - For example, if two triangles overlap the same pixel(s) in the screen.

* Technically, this process is refers to as rasterization of the triangles into and image of frame buffer. Term rasterization comes from the fact that triangles are decomposed into pixels of a raster image.

** For developing the intuition; in production, it is often that portion of scene contains millions of triangles.

Rasterizer: hint on optimization

- The naive implementation is looping through all pixels in the image even small numbers of pixels may be contained within triangle.
- Solution to this problem is computing the bounding box around the projected triangle and iterating only over pixels in this bounding box*.
- <IMAGE: bounding box>
- Note: Idea of using bounding box to optimize algorithms that perform some kind of spatial search is often used in computer graphics. More general term for those are acceleration structures which are used in all stages of rendering.

* In practice, even more optimizations is performed, this is just a hint.

Storing results

- Similarly as in raytracing, we aim to obtain an image using rasterization-based rendering.
- Once rasterization is performed, shading takes place for found pixels and those are stored in so called **frame-buffer**
- Frame-buffer is 2D array of colors that has the size of the image.
- Frame-buffer is initialized before rendering (e.g., setting all colors to black), pixels that overlap the triangle (rasterization) are recorded as colors in framebuffer (shading).
- When rendering is done, frame-buffer will contain the image of the scene visible from camera.

Rasterization: hint on visibility

- As rasterizuation is solving the visibility problem, it needs to determine which surfaces of 3D objects are visible from camera.
- Often problem is that more than one triangle may overlap the same pixel in the image.
- Rasterization commonly employs method called **z-buffer or depth buffer.**
  - Similarly as frame buffer, this is another 2D array with same dimensions as image but instead of colors it contains array of floating point numbers.
  - Before rendering, z-buffer is initialized to a very large number.
  - When pixel overlaps the triangle, we read value stored in z-buffer at that pixel and use it for determining the visibility. When pixel Pi overlaps triangle Ti distance from camera to Ti is used and compared to depth buffer at pixel Pi and updated. When the same pixel Pi overlaps triangle Tj then again distance from camera to Tj is compared to depth buffer. If the distance is smaller then Tj is visible otherwise Ti is visible.
  - Z-buffer stores the distance of each pixel to the nearest object in the scene.

Raytracing vs rasterization

- Both algorithms are solving visibility
- Both are in principle simple
- In ray-tracing, computing ray is easy but computing intersection of ray with object is complex
- In rasterization, vertices of triangles are projected which is simple and fast as well as finding pixels covered by those triangles.
- When it comes to shading, ray-tracing-based approaches have more information to work with and are more intuitive way of producing advanced effect inherently.

Application of rasterization

- Rasterization is very well suited for GPU
- Rasterization is commonly employed rendering technique on GPU - a graphics rendering pipeline which is the term used in real-time rendering.
- NOTE: Rasterization is only one part of graphics rendering pipeline (used for visibility calculation)

Practical note: rasterization-based renderer

- Rasterization-based rendering technique is deeply integrated in GPU hardware for rendering. Raytracing-based rendering is purely implemented on CPU.
- Rasterization-based rendering can also be completely implemented on CPU. For learning 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/)

Note: evolution of graphics API and hardware

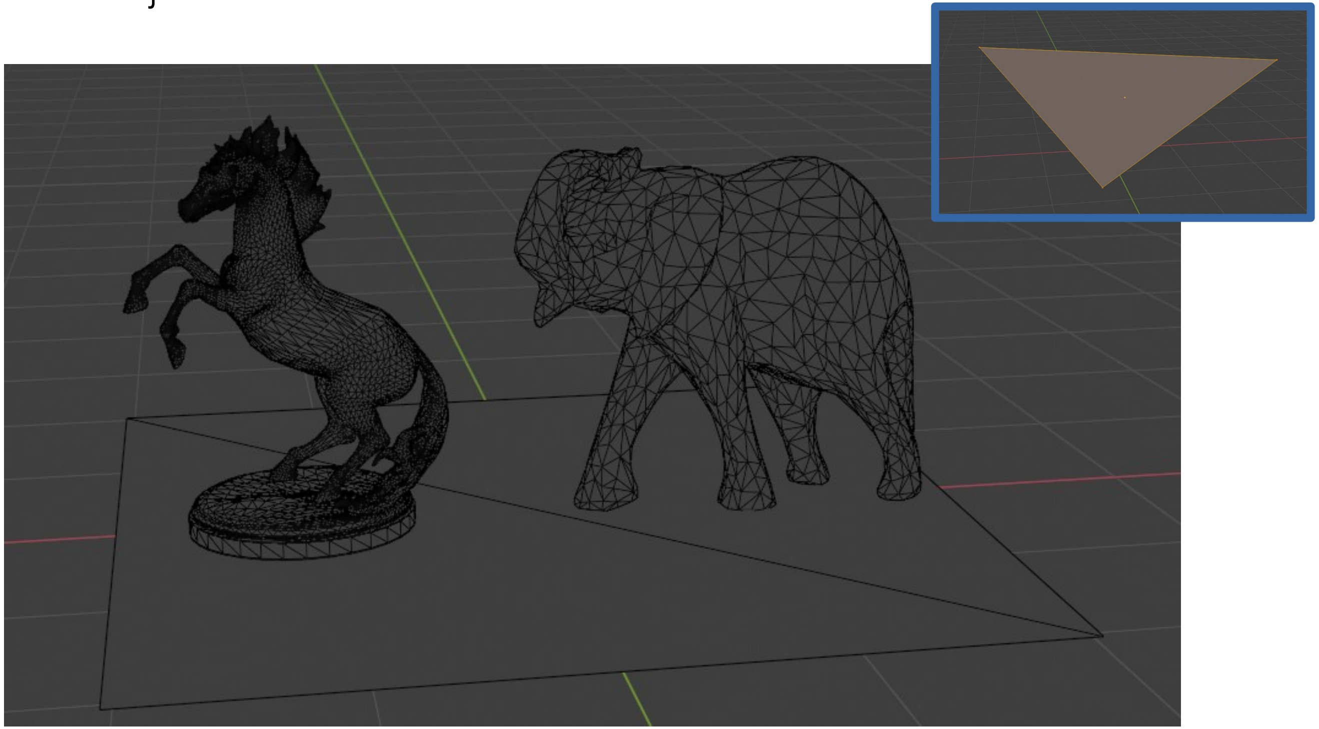
- RTR fig 3.5

Graphics rendering pipeline

Based on Real-time rendering book.

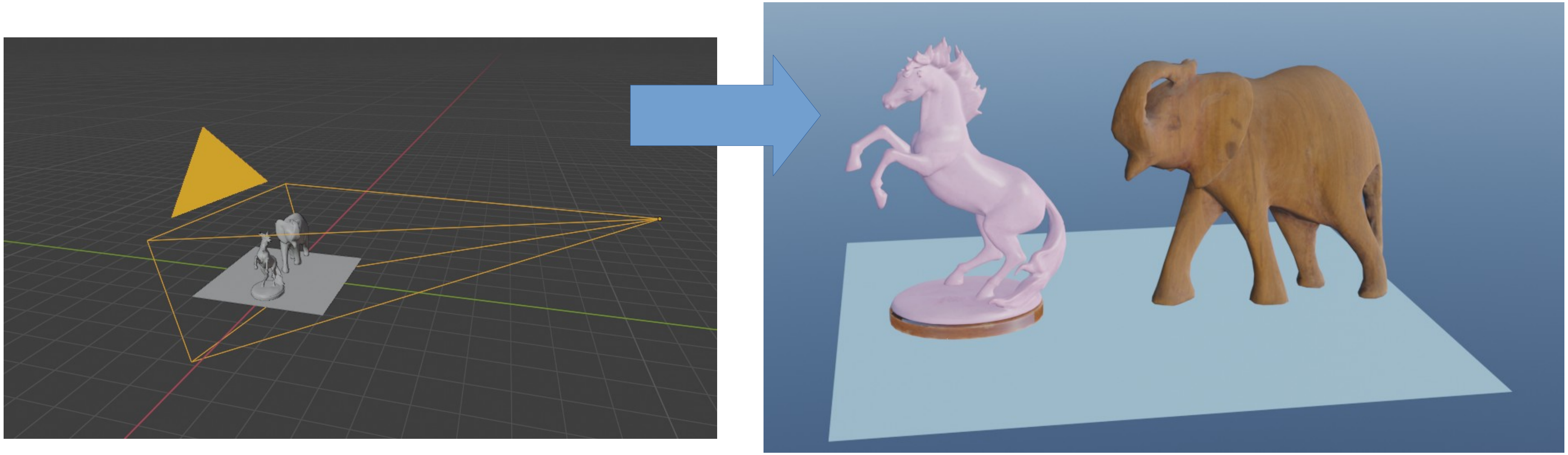
A intro note

- For graphics rendering pipeline, we assume that all objects in 3D scene, that is, their shape is triangulated mesh.
- When describing graphics rendering pipeline, we will then focus on one triangle or one vertex – but keep in mind that this is also done for all triangles for all objects/models in 3D scene



Graphics rendering pipeline

- Main function of graphics rendering pipeline (shortly pipeline): render a 2D image from 3D scene (objects, lights, cameras)
  - Underlying tool for real-time rendering
- Rendering:
  - Visibility calculation (which objects are visible from camera)
  - Shading calculation (light-matter calculation and light transport)



Virtual camera frustum and 3D objects.

Visibility: locations and shapes are determined by object geometry and camera placement.

Shading: appearance of objects detpends on materials, lights, textures, etc.

Graphics rendering pipeline

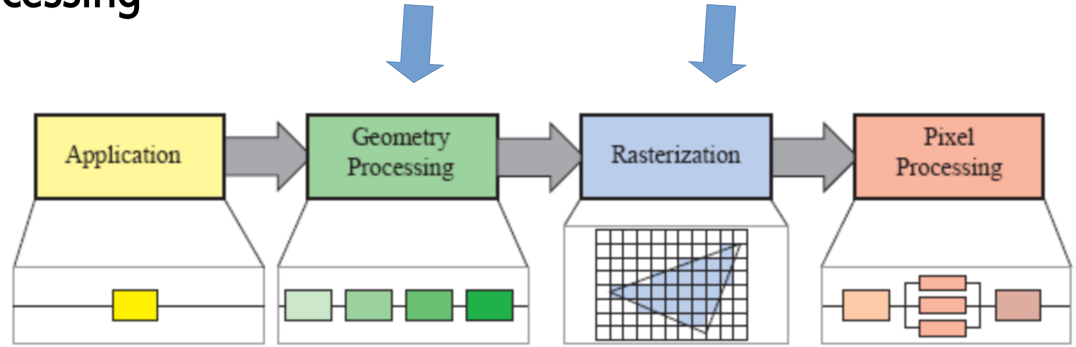
- Graphics rendering pipeline is decomposed in smaller steps or stages:
  - Each stage performs part of larger task
  - Input of any given stage depends on the output of previous stage
  - Sequence of stages forms rendering pipeline
  - Pipeline stages, although working in parallel, are stalled until 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:
  - **Application**
  - **Geometry processing**
  - **Rasterization**
  - **Pixel processing**

May consists of several substages.

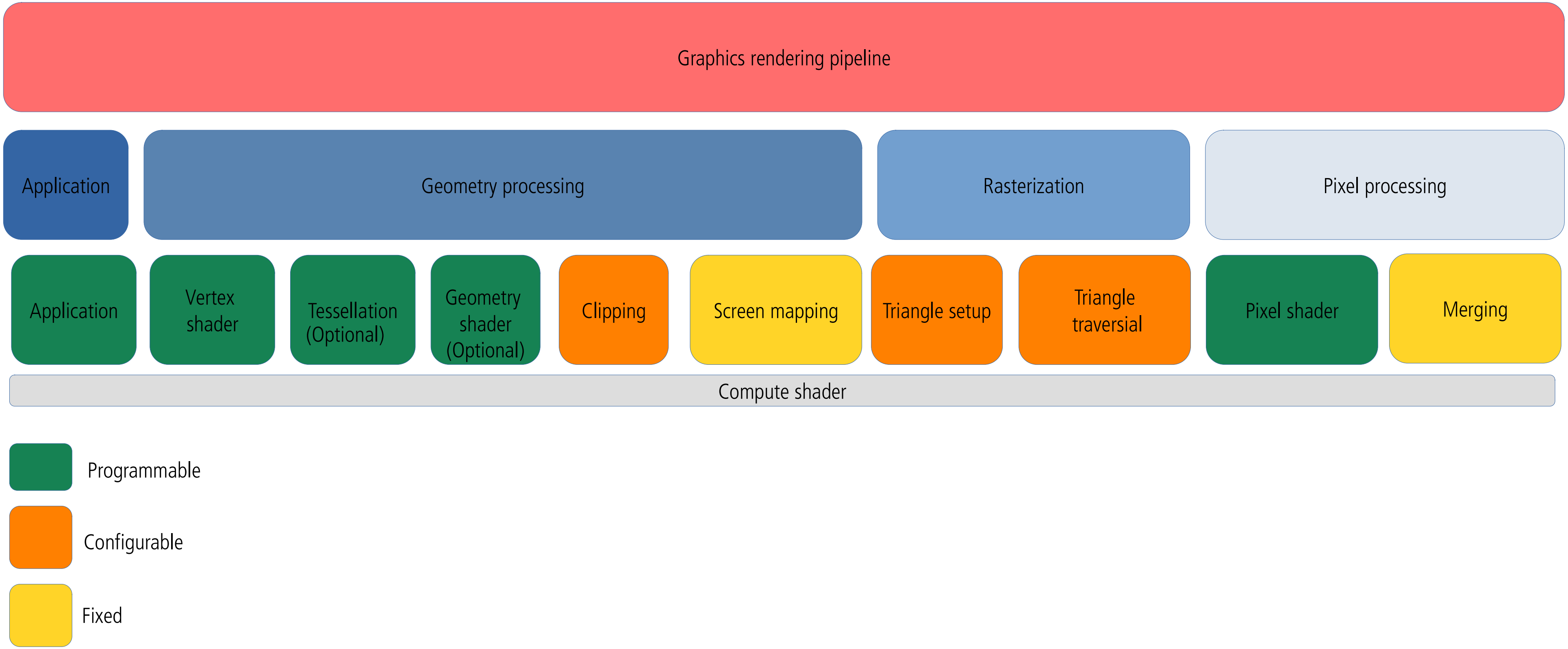
May be partly parallelized



Certain stages are fixed, some are configurable to certain extent and some are fully programmable. Trend is towards programmability and flexibility.

https://www.realtimerendering.com/

Graphics rendering pipeline overview



Graphics rendering pipeline

Application

Geometry processing

Rasterization

Pixel processing

Application

Vertex shader

Tessellation (Optional)

Geometry shader (Optional)

Clipping

Screen mapping

Triangle setup

Triangle traversial

Pixel shader

Merging

Compute shader

🟩 Programmable

🟧 Configurable

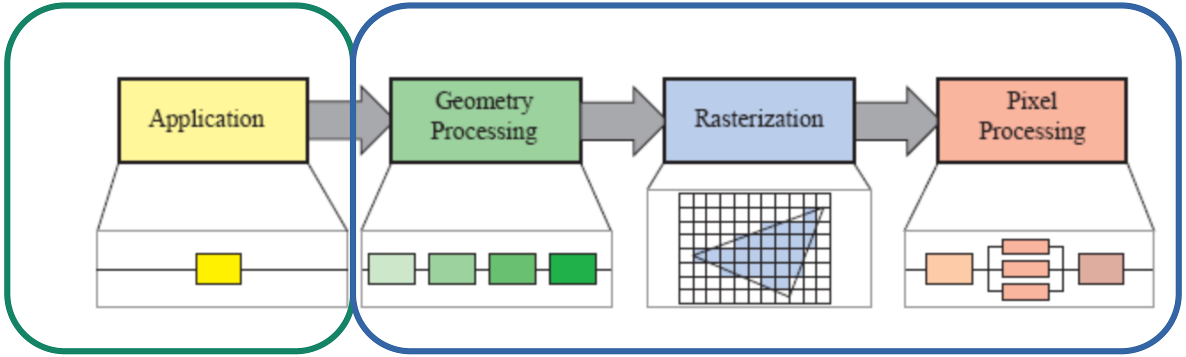
🟨 Fixed

A note on pipeline abstraction

- **Functional stages –** task to be performed but not how
- **Implemented stages –** how are functional stages implemented in hardware and exposed to the user as API.
- **Logical model of GPU** – exposed to a programmer by API. Physical model is up to the hardware vendor.

Pipeline: CPU and GPU

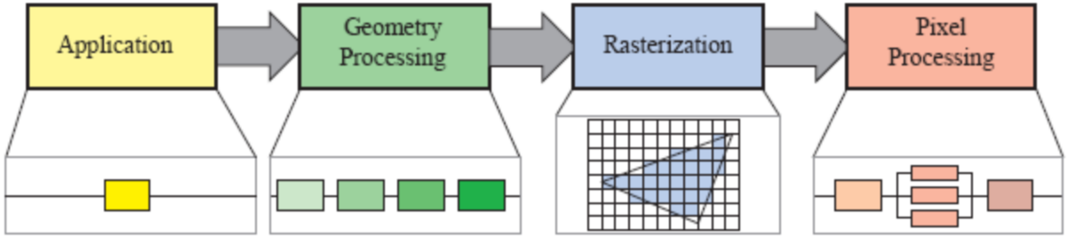
- CPU implements application stage.
  - CPUs are optimized for various data structures and large code bases, they can have multiple cores but in mostly serial fashion (SIMD processing is exception)
- GPU* implements conceptual geometry processing, rasterization and pixel processing stage.
  - GPUs are dedicated to 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**.



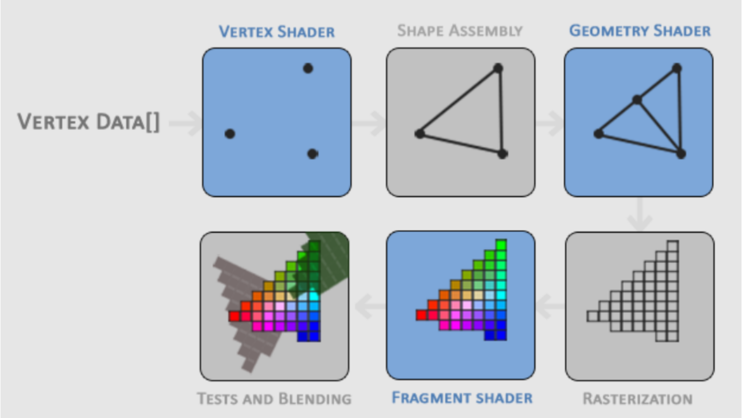
* GPU – graphics processing unit, term coined by NVIDIA to differentiate GeForce 256 from previous rasterization chips. From then on, this term is still used.

** Memory access and transfer is huge topic and efficient handling of data transfer from CPU memory to GPU memory is important for efficient rendering. The term latency describes how much processor must wait for data access.

Pipeline: speed run



- Application stage
  - Driven by application implemented in software running on CPU
  - e.g., for modeling tool application, user input is handled on application stage
- Geometry processing
  - Geometry handling (triangulated mesh): transformations, projections; what, how and where it is drawn
  - Implemented on GPU that contains many programmable shader cores
- Rasterization
  - Takes 3 vertices which form a triangle, finds all pixels inside triangle and forwards to next phase
  - Fixed implementation on GPU
- Pixel processing
  - Shading operation per pixel: calculating color and depth testing → programmable GPU shading cores
  - Per-pixel operations, e.g., blending new and old pixel color → implemented on GPU

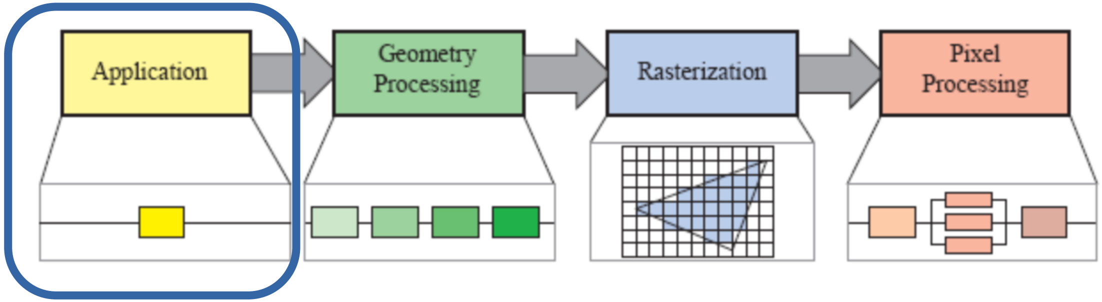


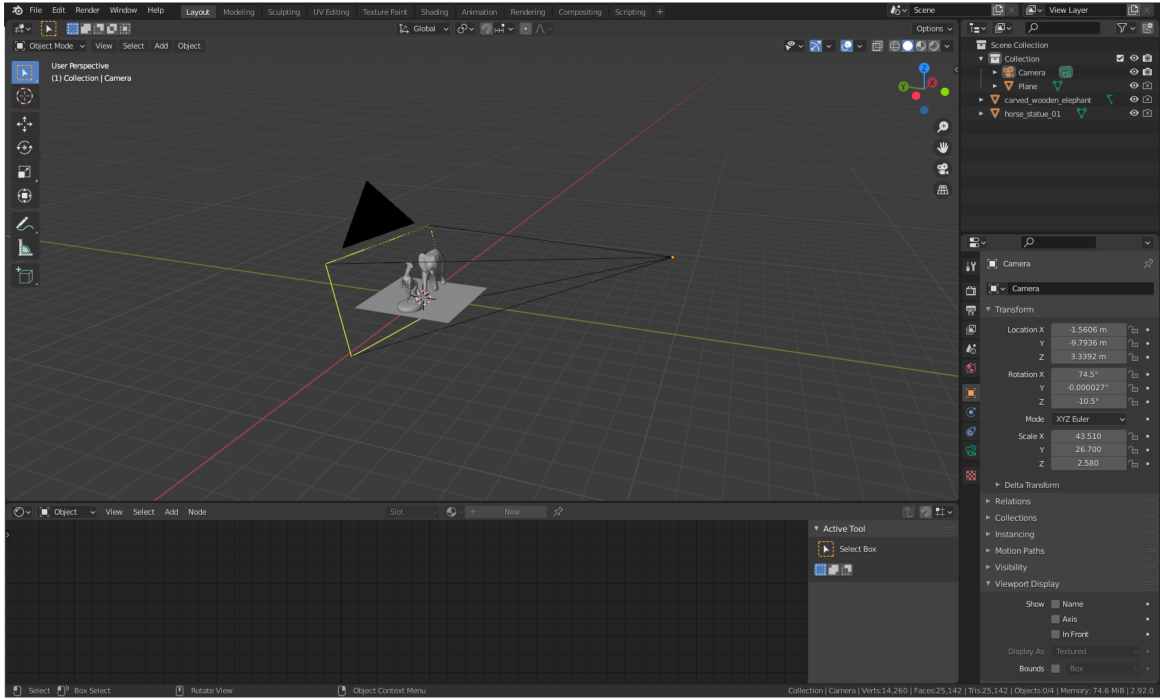
https://learnopengl.com/Getting-started/Hello-Triangle

1. Application stage

1. Application stage

- Driven by application (e.g., modeling tool) and typically implemented on CPU (optionally on multiple threads).
  - Developer has full control over what happens in this stage and how it is implemented





Camera

Transform

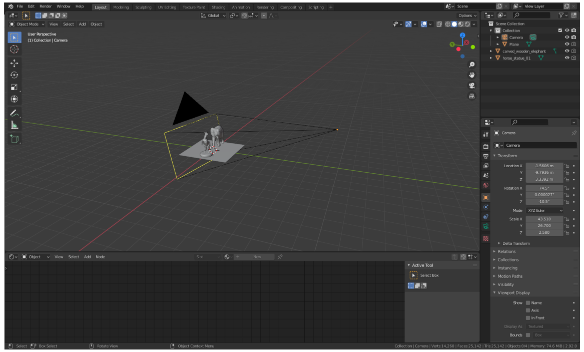
Location X	-1.5606 m
Y	-9.7936 m
Z	3.3392 m
Rotation X	74.5°
Y	-0.000027°
Z	-10.5°
Mode	XYZ Euler
Scale X	43.510
Y	26.700
Z	2.580

Delta Transform
Relations
Collections
Instancing
Motion Paths
Visibility
Viewport Display

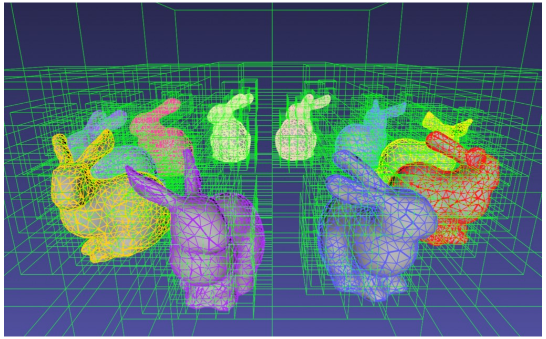
Show □ Name
□ Axis
□ In Front
Display As Textured
Bounds □ Box

Collection | Camera | Verts:14,260 | Faces:25,142 | Tris:25,142 | Objects:0/4 | Memory: 74.6 MiB | 2.92.0

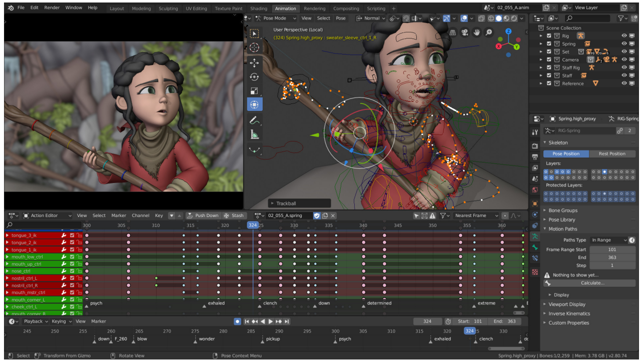
1. Application stage



- 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)
  - Composition
  - Other tasks depending on application which subsequent stages of pipeline can not handle



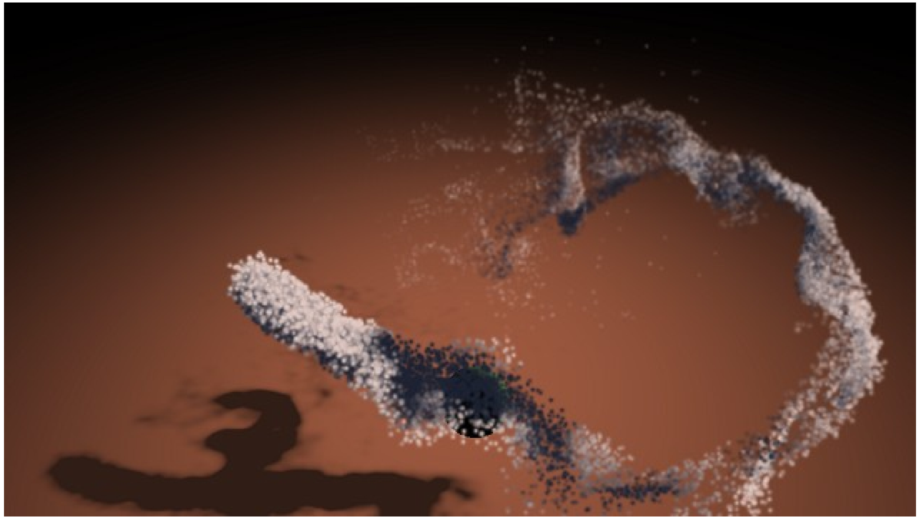
https://www.kitware.com/octree-collision-imstk/



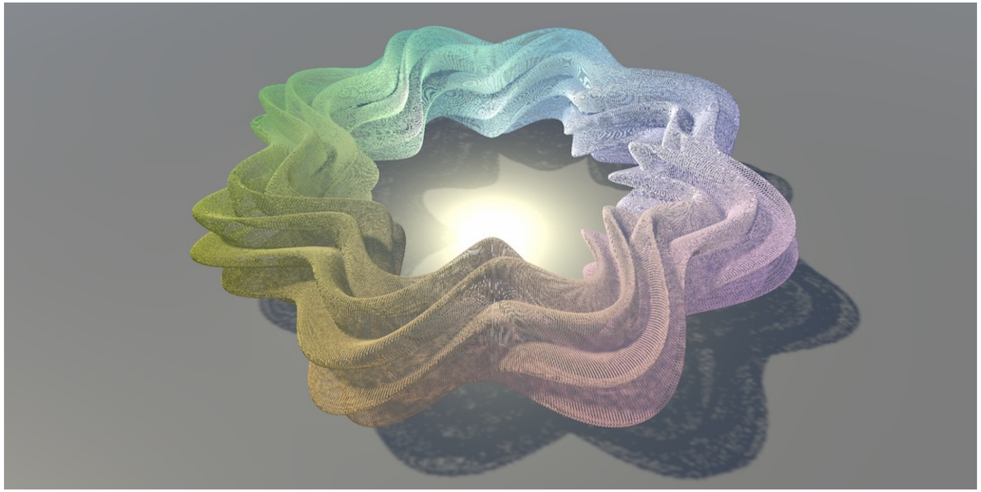
https://www.blender.org/features/

1. Application stage

- Some work on application stage can be sent to GPU for processing – **compute shader**.
  - Treat GPU as highly parallel general processor



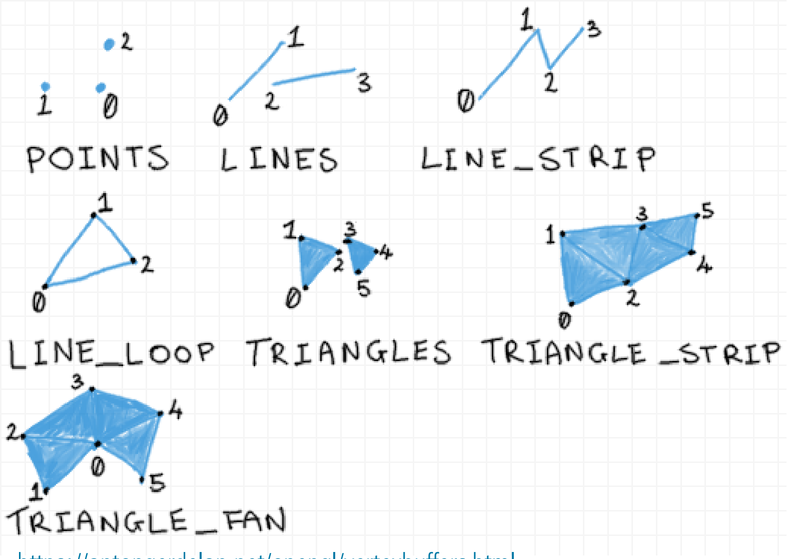
https://arm-software.github.io/opengl-es-sdk-for-android/compute_particles.html



https://catlikecoding.com/unity/tutorials/basics/compute-shaders/

1. Application stage

- Application phase outputs geometry to be rendered – **rendering primitives:** points, lines and triangles
- Efficiency of this stage is propagated to further stage: geometry processing
  - e.g., amount of geometry (triangles) sent to GPU

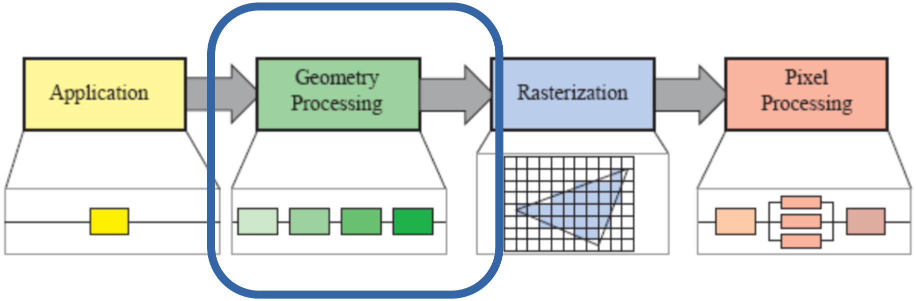


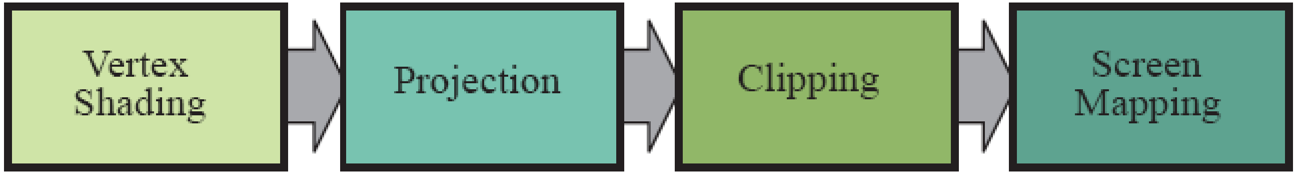
https://antongerdelan.net/opengl/vertexbuffers.html

2. Geometry processing stage

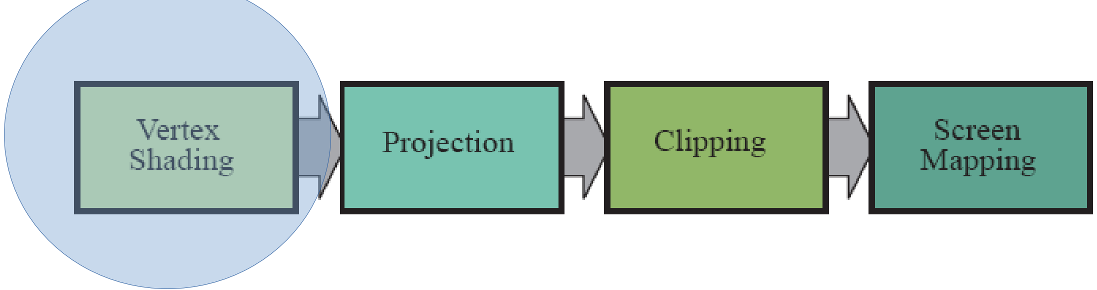
2. Geometry processing stage

- 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 (vertex shader)**
  - 2.2. **Projection**
  - 2.3. **Clipping**
  - 2.4. **Screen mapping**





2.1. Vertex shading



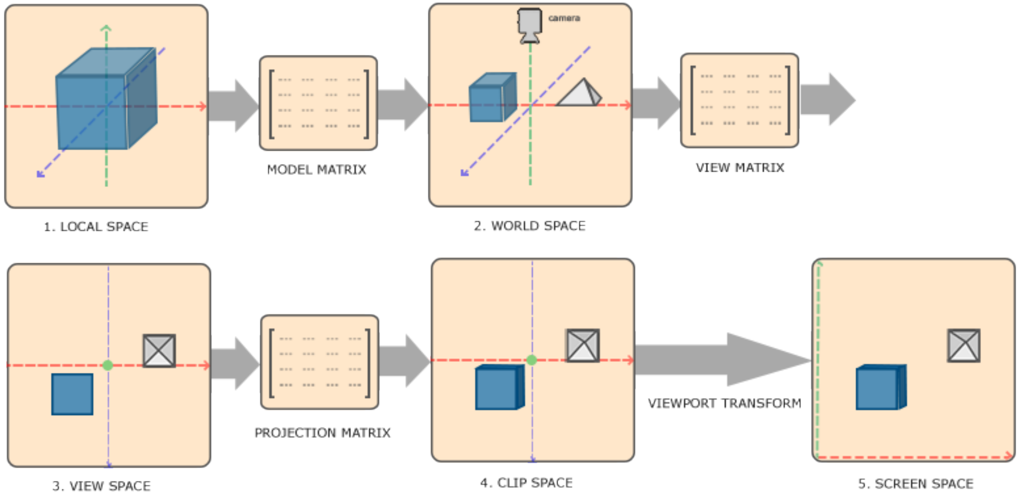
- First stage on GPU, completely under programmer's control.
- Before this stage, data manipulation from application stage is done by **input assembler**
  - For example, an object can be represented by array of positions and array of colors. Input assembler would create object's primitives (e.g., triangle) by creating vertices with positions and colors.
- First stage to process a triangle mesh. It provides a way to modify, create or ignore values associated with each triangle's vertex (e.g., color, normal, texture coordinates and position).
  - Note that normals do not have to be triangle normals rather normals of smooth shape which triangle mesh represents.
  - Note that vertex shader can not create new vertices or destroy existing ones
- Two main tasks:
  - Compute **position of vertex** given input from application stage
  - Evaluate/set-up any additional **vertex data output** desired by programmer such as normal and texture coordinates

Vertex shading applications

- An example of vertex shader task is animating objects using transformations on vertices.
  - Vertex blending – RTR 4.4
  - Morphing – RTR 4.5.
- Object generation: creating mesh once and then deforming it by vertex shader
- Animating characters using skinning and morphing
- Procedural deformations such as cloth or water
- Instancing objects: copying same object in different positions in the scene.
- Particle creation: using degenerate (no area) mesh down the pipeline and using those as positions where instancing takes place
- VFX: lens distort, heat haze, water ripples, page curls
- Terrain modeling by applying height fields given by texture

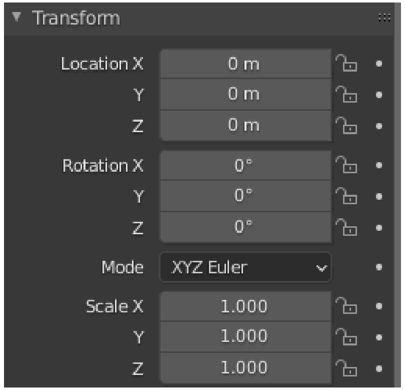
Vertex shading: computing vertex position

- **Vertex positions** of a model is 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

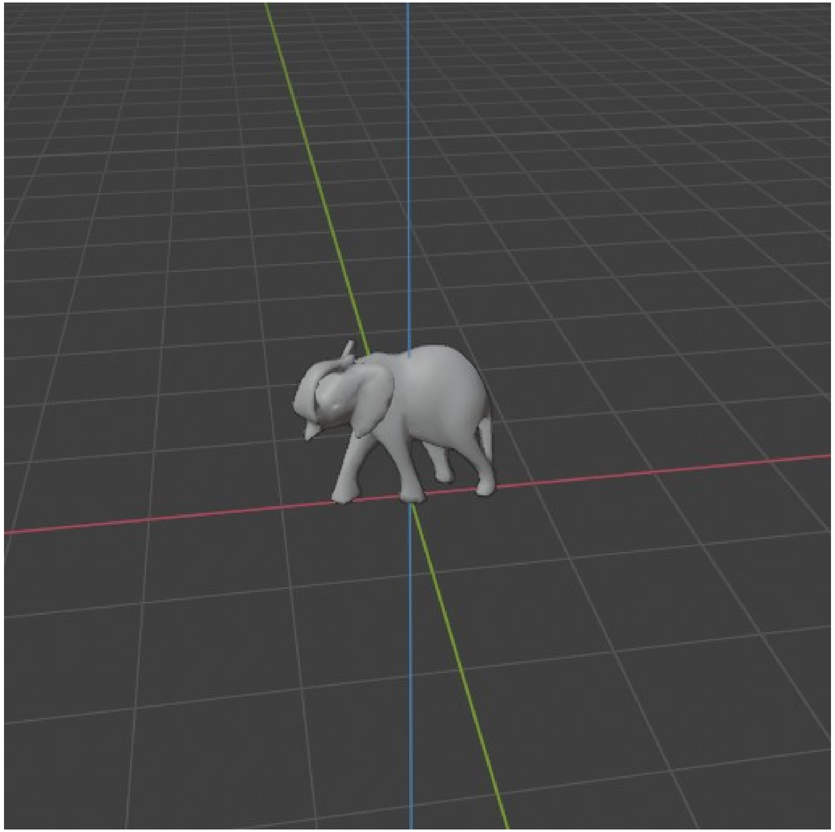
- Originally, model (vertex positions) given to vertex shading stage is in its own space – **model space**.
  - Model space → model has not been transformed at all.
- Each model can be associated with **model transform –** for positioning and orienting
- Model transform is applied on model's vertices and normals.
  - Coordinates of a object are also called model **coordinates**.



▾ Transform

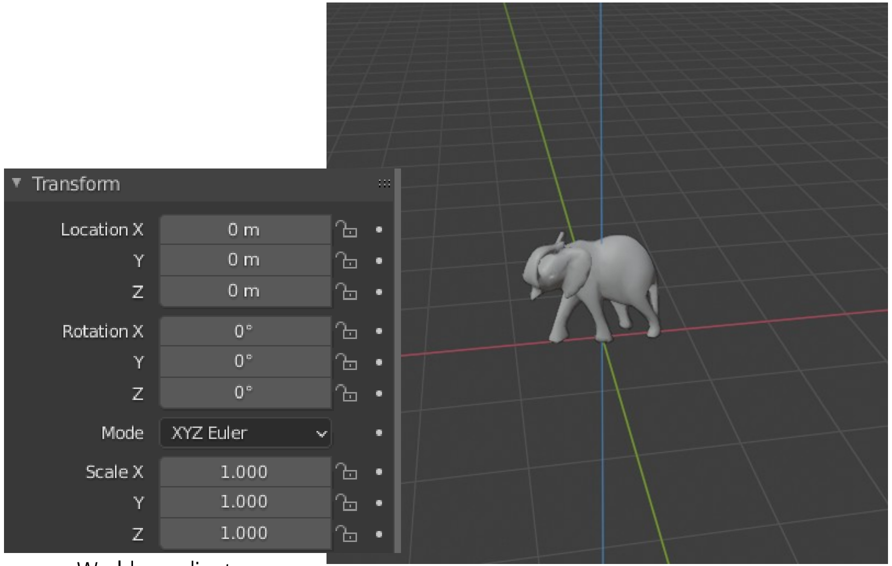
Location X	0 m
Y	0 m
Z	0 m
Rotation X	0°
Y	0°
Z	0°
Mode	XYZ Euler
Scale X	1.000
Y	1.000
Z	1.000

World coordinates



World space

- After model transform has been applied on model coordiantes, the model is said to be in **world coordinates** or **world space**.
  - World space = scene
- World space is unique and after all models have been transformed with their respective model transforms, they all exist in the same space – world space.
- Model transform is defined on application stage and performed by **vertex shader**.

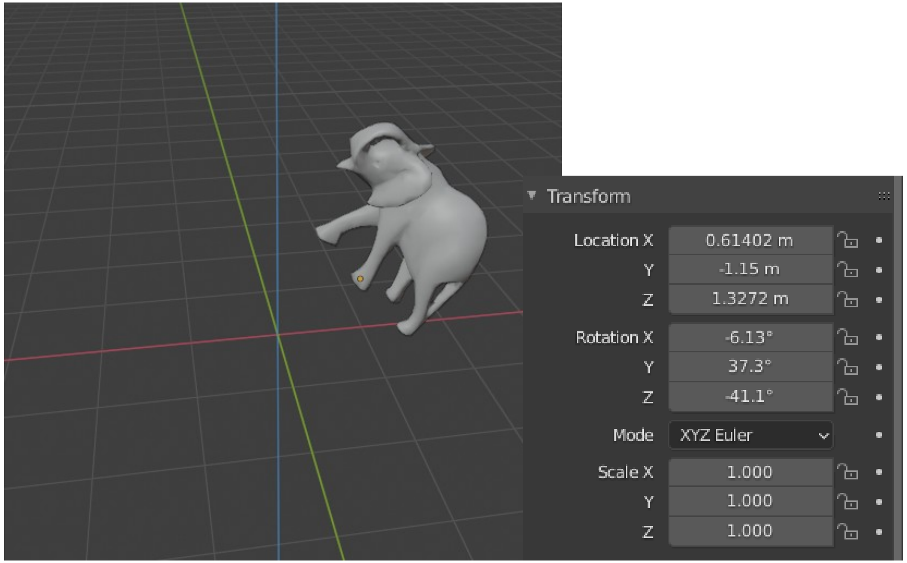


Transform

Location	X	0 m
	Y	0 m
	Z	0 m
Rotation	X	0°
	Y	0°
	Z	0°
Mode		XYZ Euler
Scale	X	1.000
	Y	1.000
	Z	1.000

World coordinates

Model transform



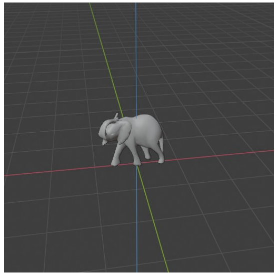
Transform

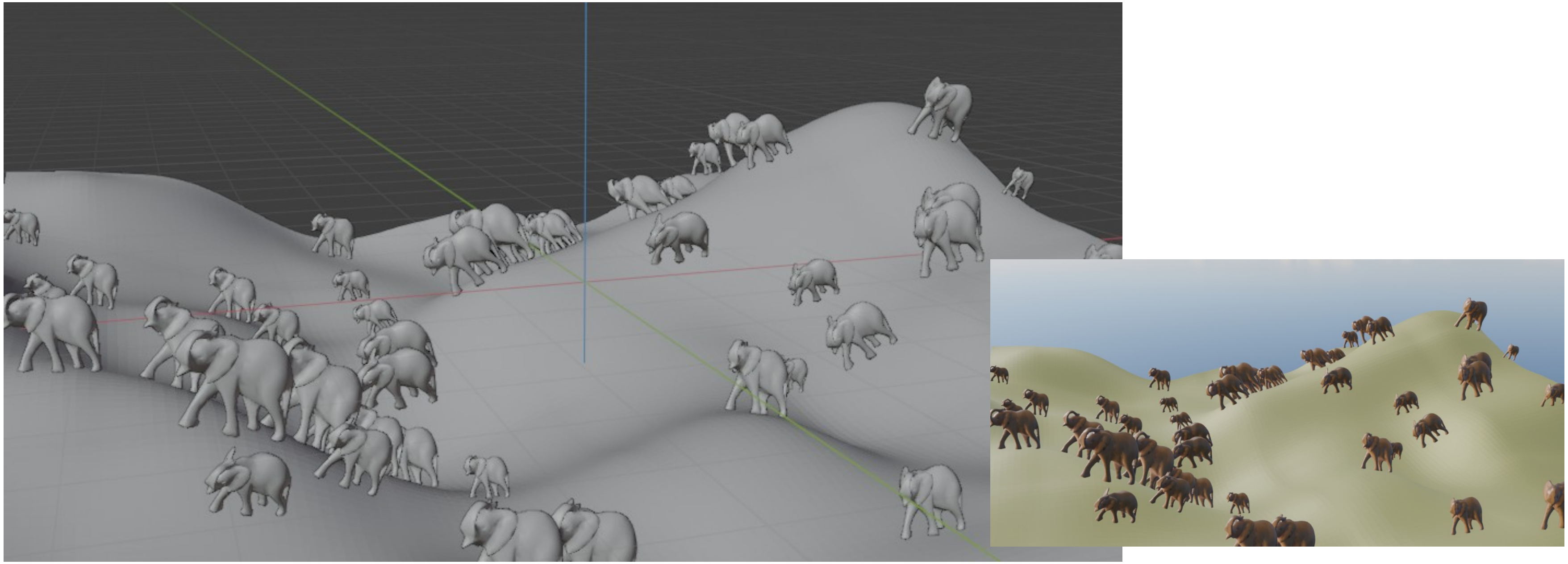
Location	X	0.61402 m
	Y	-1.15 m
	Z	1.3272 m
Rotation	X	-6.13°
	Y	37.3°
	Z	-41.1°
Mode		XYZ Euler
Scale	X	1.000
	Y	1.000
	Z	1.000

World coordinates

Model transform

- 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.





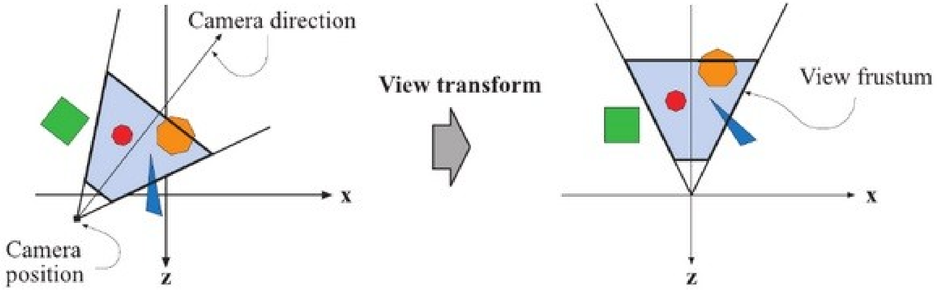
Instancing

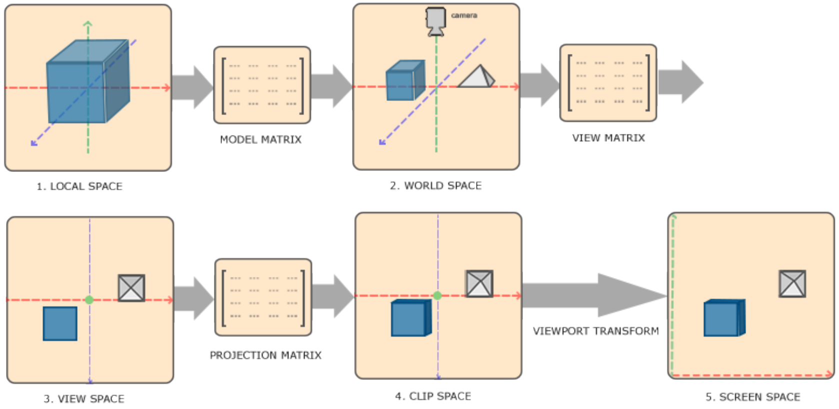
- Very much used for environment art and object placing repeating objects in 3D scene

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

Camera (view, eye) space

- Only objects visible from camera are rendered.
- Camera has a **location and orientation in world space.**
- 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
  - Result: 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.





https://www.realtimerendering.com/

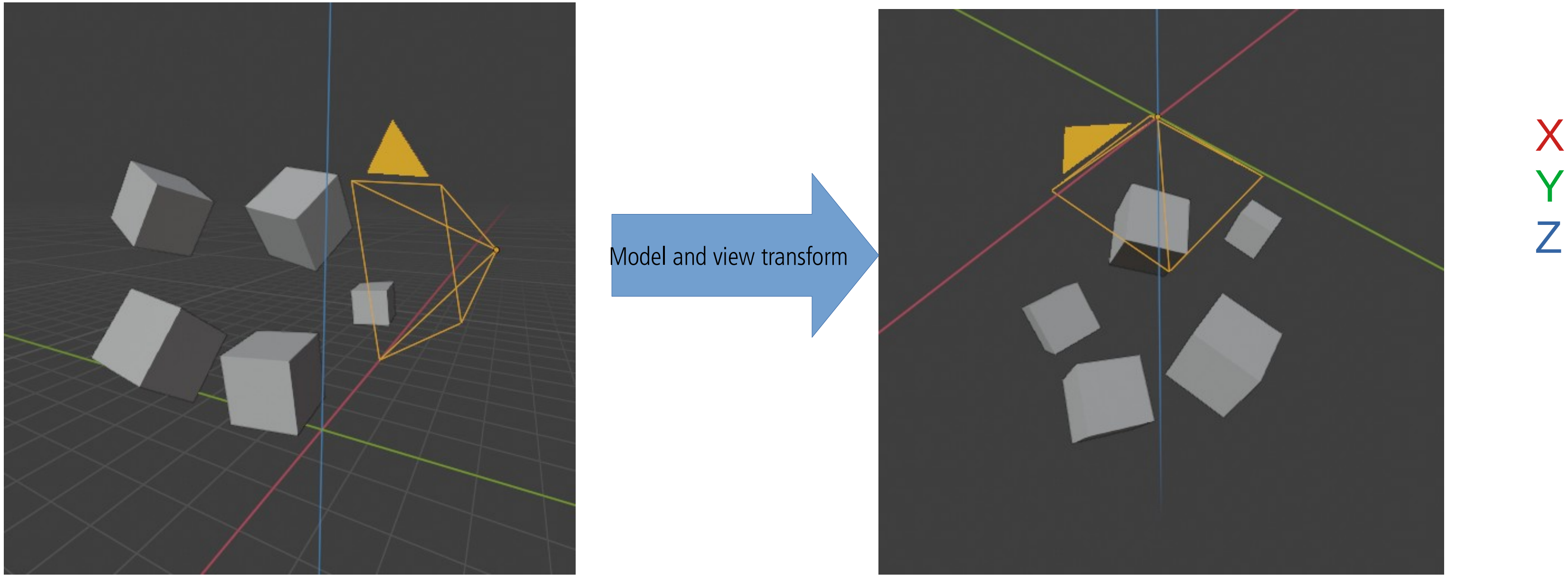
* This convention is called negative z axis convention. Another convention is positive z axis convention. All are fine but must be consistently used when decided. Actual position and direction after view transform are dependent on underlying API.

Digression: camera

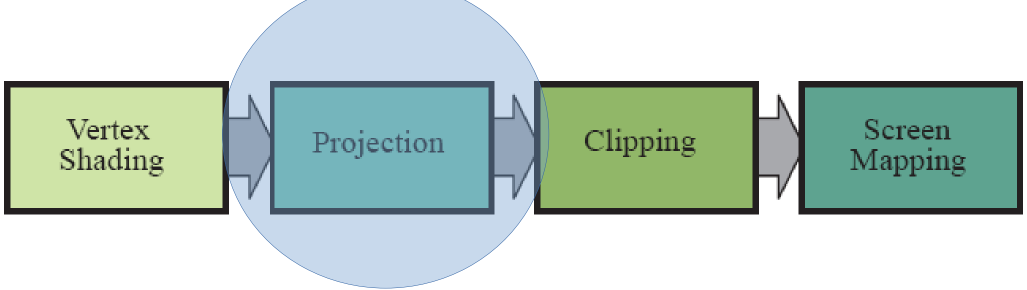
- Similarly as object vertices are defined on application stage, camera properties are also
- Camera properties can be described with location and transformation matrices which are sent further down the pipeline

Camera (view, eye) space

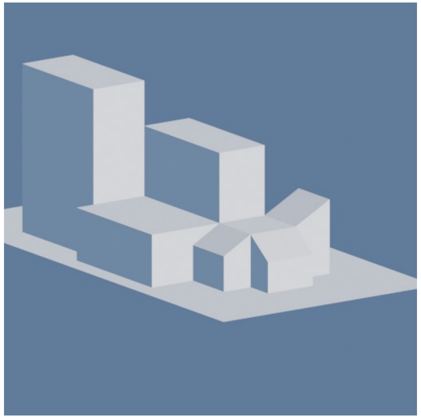
- After **view** transform, the model is said to be in **camera (view or eye) space.**
- Model and view transform can be implemented as one 4x4 matrix for efficient multiplication with vertex and normal vectors.
  - Note: programmer has full control over how the position and normals are computed.

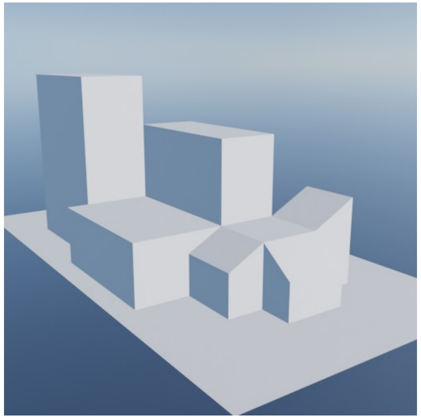


Projection



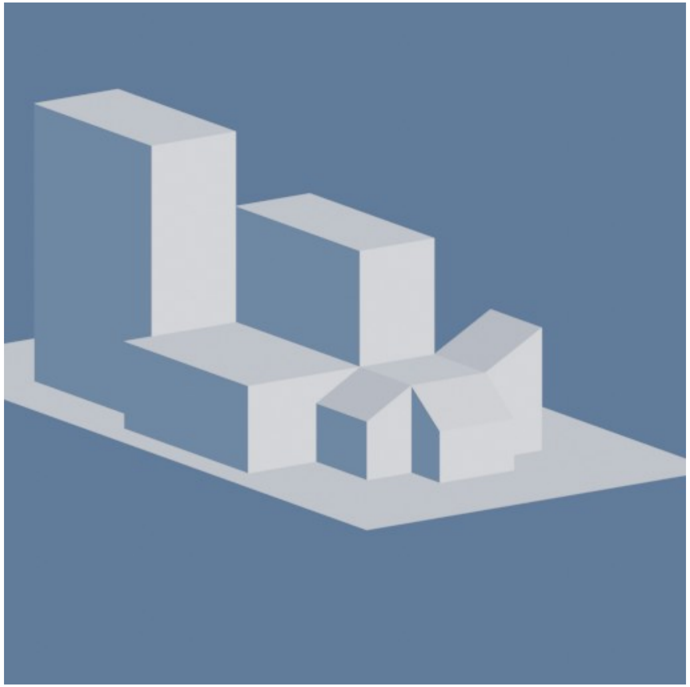
- After models are transformed to camera space, a projection is applied.
- Two commonly used projection methods:
  - **Orthographic**
  - **Perspective**
- Projection is represented as matrix and can be combined with other geometry transforms: model and camera.
  - Projection matrix is defined in application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

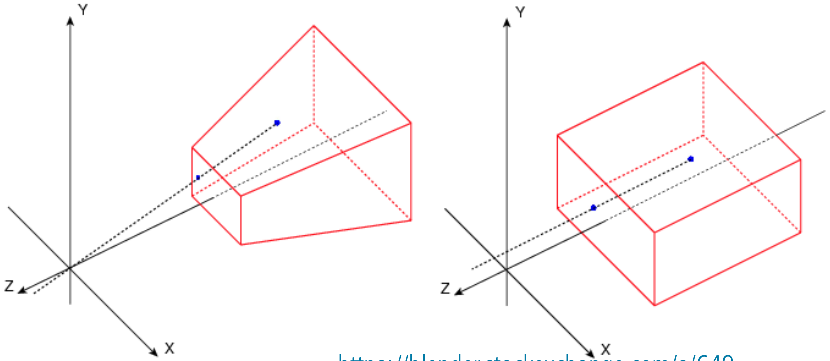




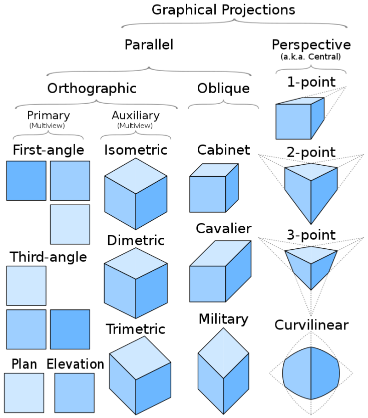
Orthographic projection

- 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!

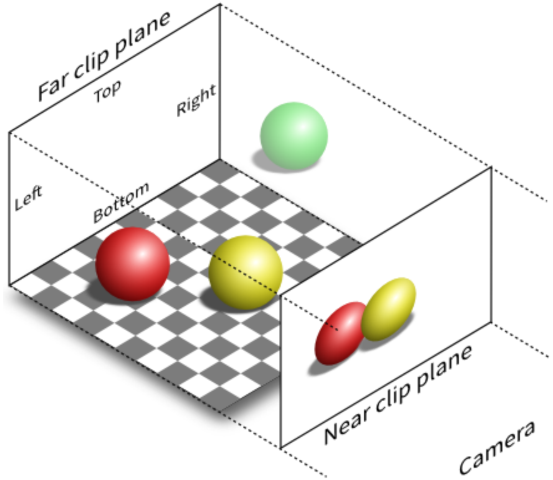
$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad Pv = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ 0 \\ 1 \end{bmatrix}$$

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**

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

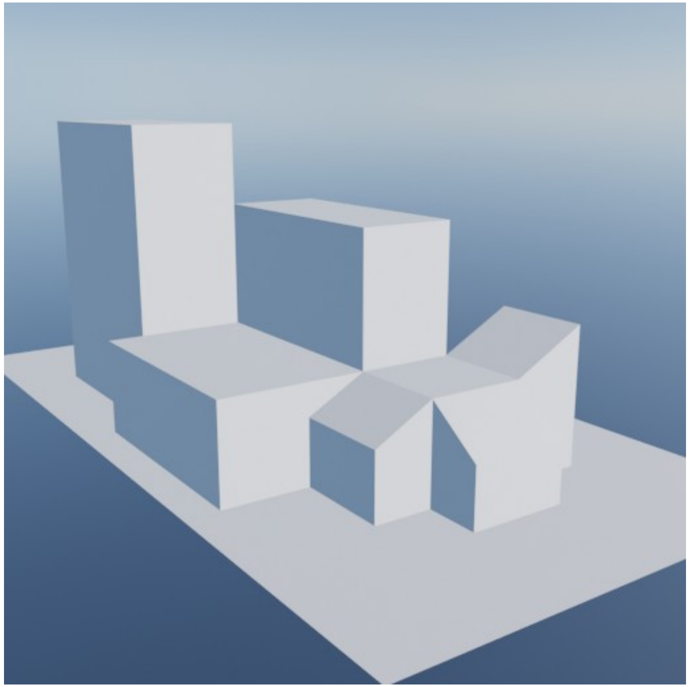
* DirectX has bounds (-1,-1,0) and (1,1,1)

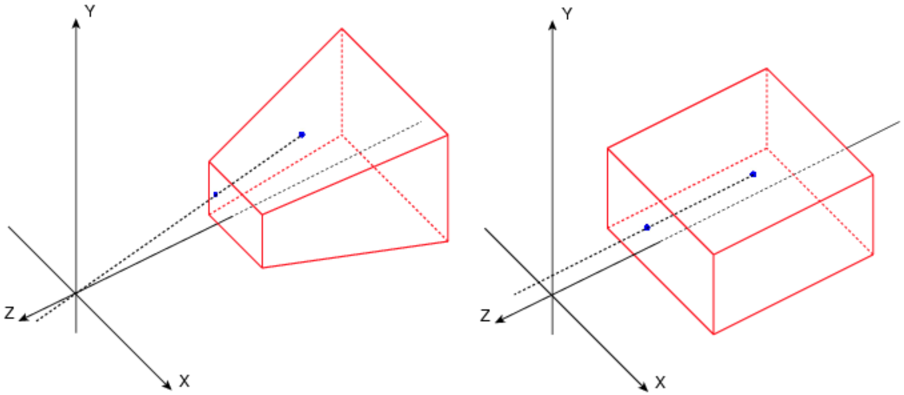


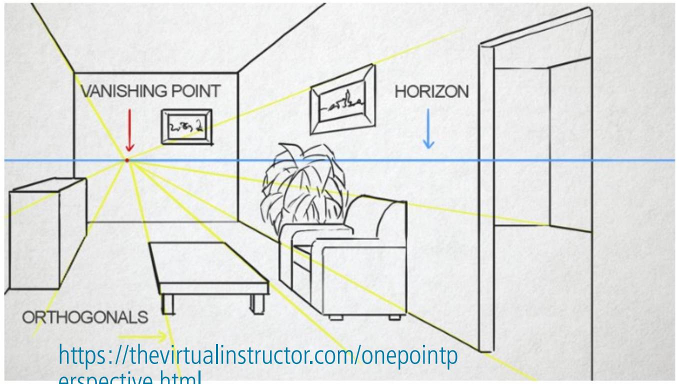
https://stackoverflow.co

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

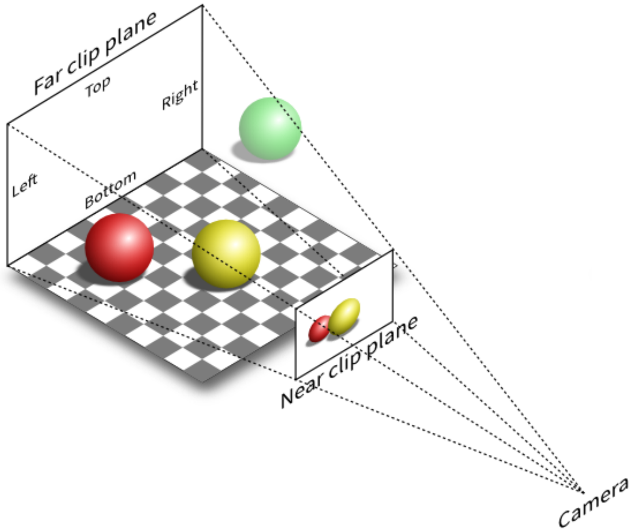
- **Simple perspective projection** is projecting points on a plane `z = 1` → information on depth is lost!
- After multiplication with perspective matrix, homogeneous coordinate w_c will be equal to vertex coordinate z. Other stays the same
- Perspective divide gives vertex on a plane

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

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

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)
- Construction of this matrix is supported by various libraries and supported in OpenGL Utility Library (GLU) as `gluPerspective()` https://registry.khronos.org/OpenGL-Refpages/gl2.1/xhtml/gluPerspective.xml



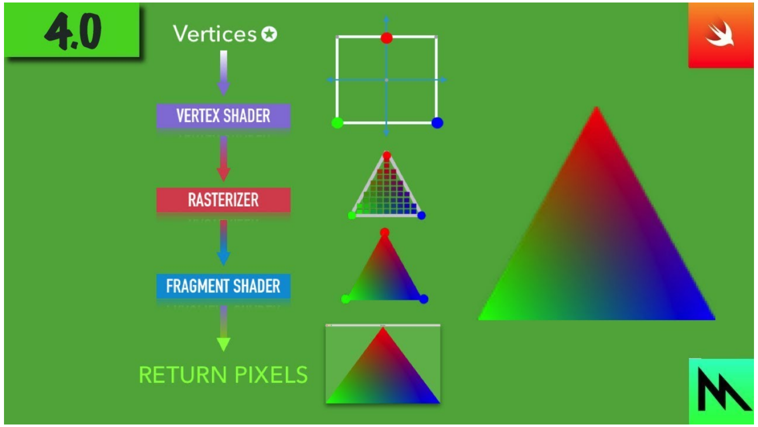
$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Projection

- After projection coordinates are in **clip space - clip coordinates**
  - These are homogeneous coordinates (division with w 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.

Vertex shading: computing additional vertex output data

- **Shading** can also be performed on this stage*
  - Computing color using on vertex data (position, normal, material) and light
  - Result is stored as color per vertex which is later interpolated and used cross triangle and pixels
  - Vertex shading is performed using vertex shader using vertex data defined on application stage.



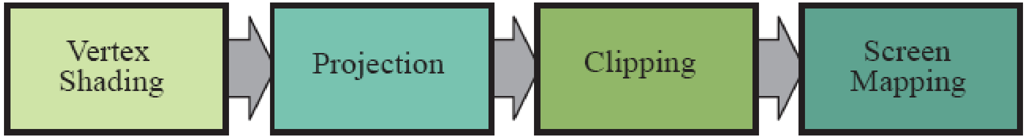
https://www.youtube.com/watch?v=F7bpcyPhiH8&ab_channel=2etime

* Used when simpler and more faster shading is needed. This is why programmable vertex processing unit was named "vertex shader" also the name: "vertex shading" performed by this stage which is kept even today. In modern GPU-s and API-s, some or all shading takes place in pixel processing stage. Vertex shading stage is more general and doesn't have to perform equation at all – depending on programmer. Vertex shader is more ge neral unit dedicated to setting up data associated with each vertex.

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.

Clipping

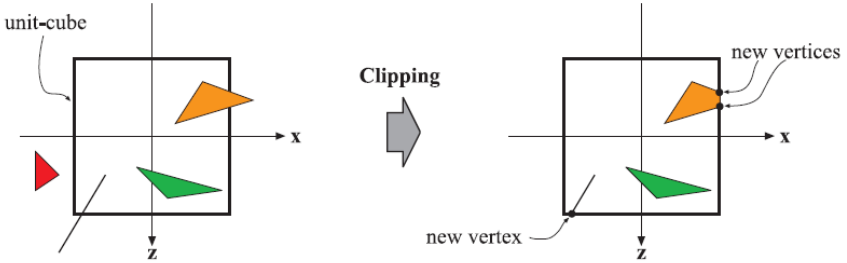

Vertex Shading

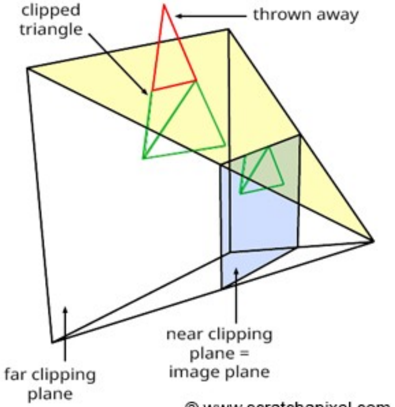
Projection

Clipping

Screen Mapping

- 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.
- Clipping uses clip coordinates resulted from projection.
  - Fourth coordinate is used for correct interpolation 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).

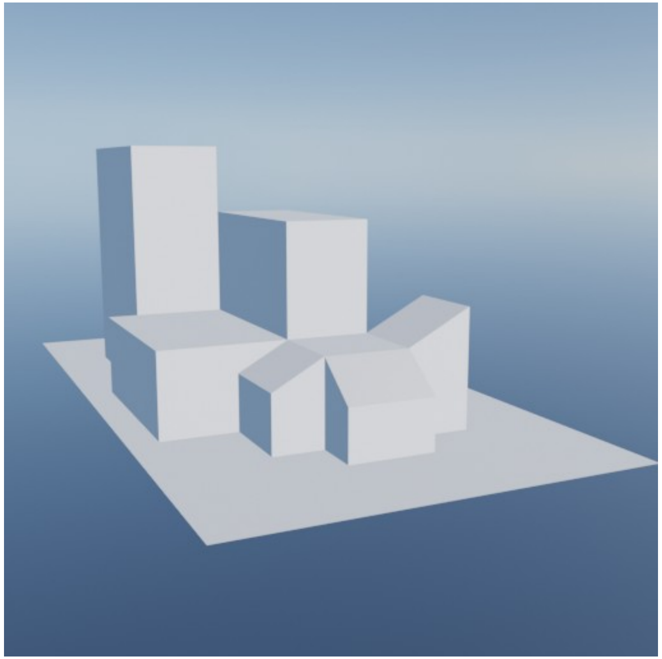


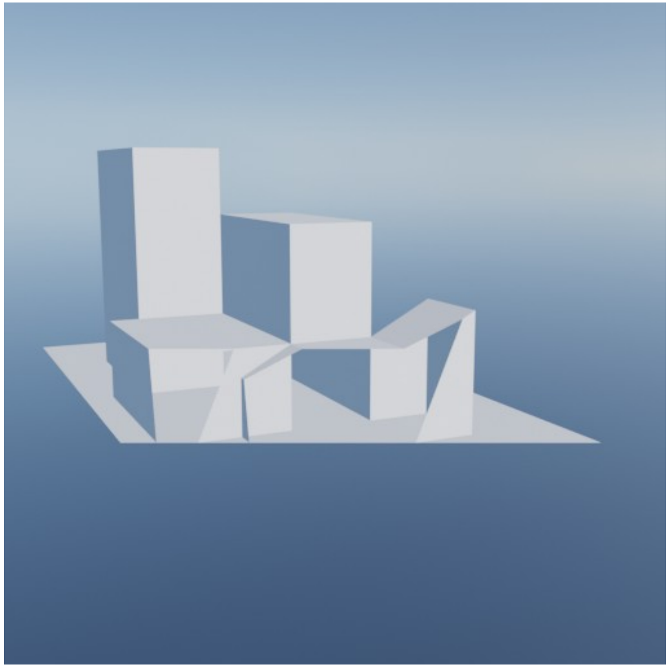


© www.scratchapixel.com

Clipping

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





Screen mapping

- Clipped primitives inside view volume are passed on **screen mapping**.
- Coordinates entering this stage are still 3D.
- y and y coordinates are transformed to form **screen coordinates**.
  - Screen mapping is translation followed by scaling to map to screen with dimensions (x1,y1) and (x2, y2).
  - <IMAGE>
- Screen coordinates with z coordinates are called **window coordinates**.
  - Z coordinates are mapped in (0,1). DX vs GL.
  - <IMAGE>
- Window coordinates are passed to the rasterizer stage.

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.
  - 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.
  - To test if triangle is overlapping a pixel, different methods exist. Point sampling may be used where only center of pixel is used for testing. Often, multiple samples per pixel are desired to evade **aliasing** problems – **multi-sampling or anti-aliasing**.
- Two functional sub-stages**:
  - 3.1.Triangle setup
  - 3.2 Triangle traversal

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

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

3.1 Rasterization: triangle setup

- Data needed for triangle traversal, interpolation and shading are computed here (e.g., differentials, edge equations)
- Programmer has no control over it.

3.2 Rasterization: triangle traversal

- Each pixel sample is checked if covered by a triangle. Finding which samples for which pixels are inside a triangle is called **triangle traversial**
- Part of the pixel that overlaps the triangle is generated and called the **fragment**
- Each triangle fragment properties are generated using data **interpolated** 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 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

- All pixels or pixel fragments that are considered inside a triangle (or other primitive) are found in previous stages. Now, computations on those are made.
- Pixel processing stage can be divided into:
  - 4.1 Pixel shading
  - 4.2 Pixel merging
- Executes program per pixel to determine its color and if color is visible (depth testing) as well as blending of pixel colors with newly computed with old color.
- Runs on GPU.

4.1 Pixel processing: pixel shading

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

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 responsible to combine (blend) fragment color produced by pixel shading stage with the color currently stored in color buffer using depth values of those fragments (also available after pixel shader stage).
  - For opaque surfaces not blending is needed: fragment color simply replace the previously stored color
  - Blending of fragment and stored color is important for transparent objects and compositing operations.
  - This stage is also called **raster operations pipeline (ROP)** or **render output unit***.
- This stage is not fully programmable, but highly configurable** for achieving various effects (e.g., transparency). It 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.

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

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

Pixel processing: 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.
- 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.

* 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.

Pixel processing: z-buffer

- Let’s recap the pipeline for now:
  - Fragment generated by rasterization
  - Fragment is then run through pixel shader
  - 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.

Pixel processing: alpha channel

- Color buffer also contains information about **alpha value – alpha channel.**
- Alpha value 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

Pixel processing: 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, we can render filled rectangle 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 some special effects.

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

Pixel merging: color blending

- As mentioned, merging stage is not programmable but highly configurable.
- 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

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).

Compute shader

- Compute shader is form of GPU computing, not locked into a location in the graphics pipeline.
  - It is like shader we discussed: it has some input data, can access buffers (e.g., texture) for input and output.
  - Using hardware this way is called GPU computing – CUDA and OpenCL are used to control the GPU as massive parallel processor.
- It is closely tied to the process of rendering graphics API: it is used alongside vertex, pixel and other shaders.
- It is used in:
  - 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
  - Replacing tessellation hull shaders.
- With compute shaders, GPU can be used more than implementing traditional graphics pipelines:
  - Training neural networks

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.

* In APIs such as OpenGL and Vulkan the term swap chain is used for this process https://en.wikipedia.org/wiki/Swap_chain

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 parameters such as position and view direction must be updated by application.
  - For each frame (CAD programs are real-time when in modeling mode) application must provide information for next – geometry stage: camera position, lighting and primitives of the model.
- Geometry processing
  - Application provided parameters of the camera which includes projection matrix. Next, application, for each object, calculates a matrix which describes location and orientation of the object.
  - Object is put in view space and optionally shading per vertices is preformed using provided light and material information.
  - Projection is applied on the object tranforming it in unit cube space that represents what the eye sees. Primitives outside of that cube are discarded. Primitives intersecting this cube are clipped.
  - Finally, vertices are mapped into the window on the screen
- 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.

Graphics rendering pipeline in practice

Code

Application of graphics rendering pipeline

- Graphics rendering pipeline is implemented on GPUs and is basis for real-time rendering
- Now, we will discuss practical application of graphics rendering pipeline.

Graphics rendering pipeline: OpenGL

- Application
  - Assimp
  - RTR: 16.4.
- Vertex shader
- Fragment shader
  - https://learnopengl.com/Lighting/Basic-Lighting
- Output

Graphics rendering pipeline in practice

Production

- Show graphics pipeline rendering example in Blender EEVEE