Games and Simulation

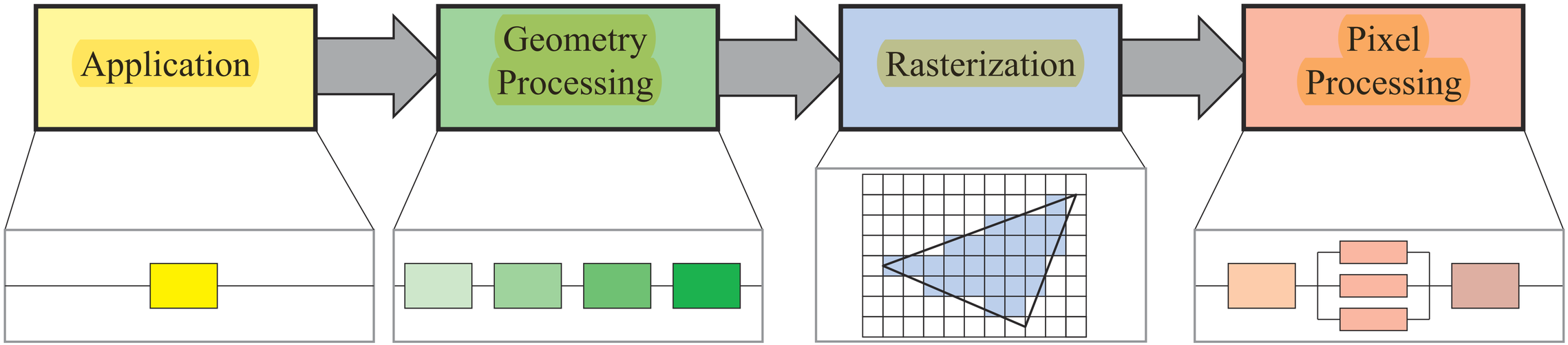
2021-2022

Fernando Birra
Rui Nóbrega

Graphics Rendering Pipeline

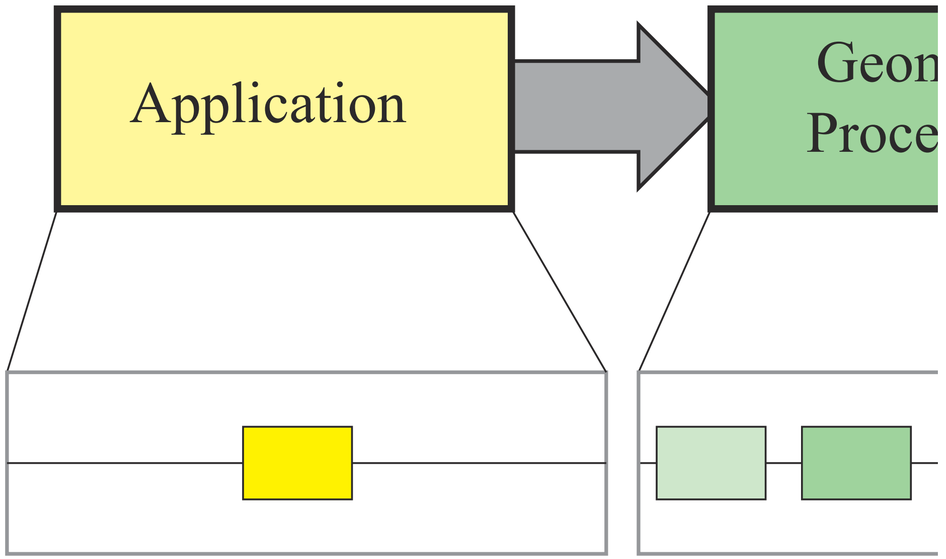
Fernando Birra
Rui Nóbrega

Architecture



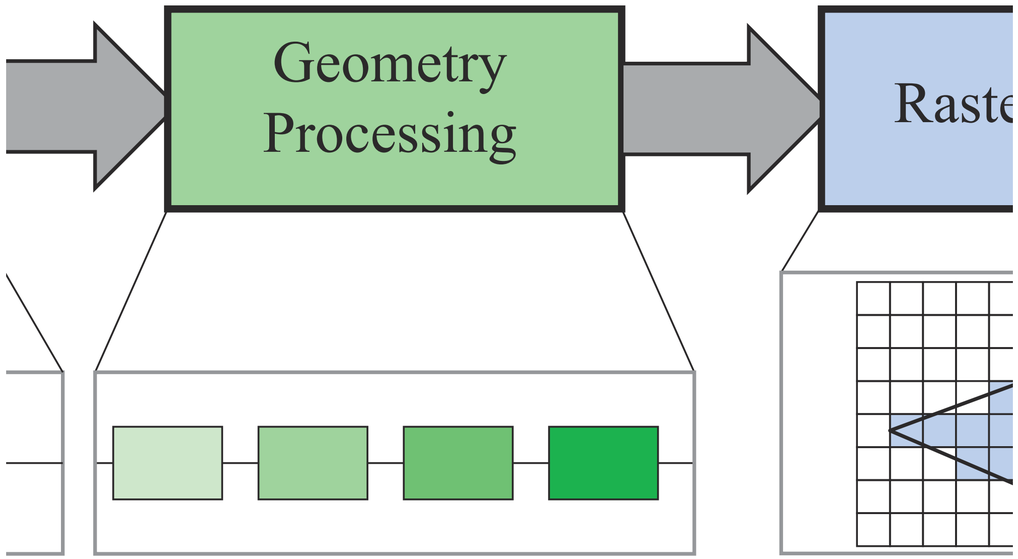
Application

- Driven by the application and is typically implemented in software, on the CPU
- Several cores can be used to handle different tasks in parallel such as collision detection, acceleration algorithms, animation, physics simulation, handle input, …
- The developer has full control of this stage
- What is done here can affect the performance of later stages. For instance, a better geometry pruning algorithm may significantly reduce the number of primitives to be rendered
- The GPU can help in this stage by using “compute shaders”
- The final goal of this stage is to feed geometry data to the next stage (rendering primitives)

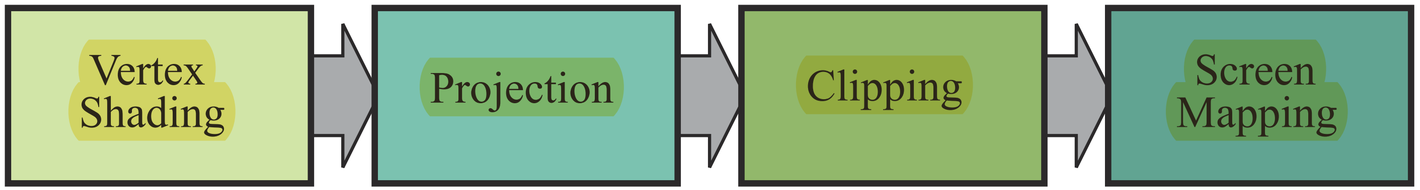


Geometry Processing

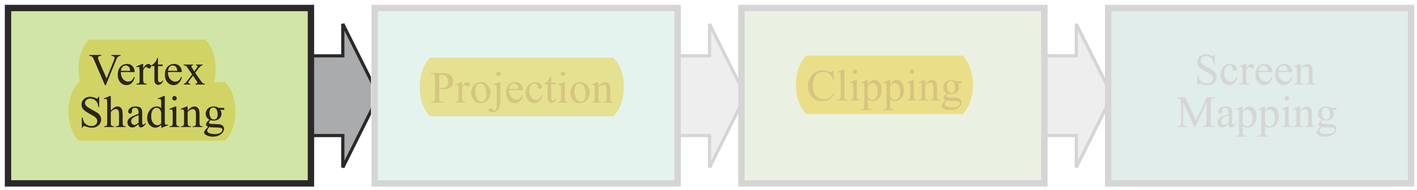
- Responsible for applying transformations, perform projection and additional handling of geometry
- Determines what should be drawn, how it should be drawn and where it should be drawn
- It is generally executed in the GPU using many cores and fixed-operation hardware



Geometry Processing

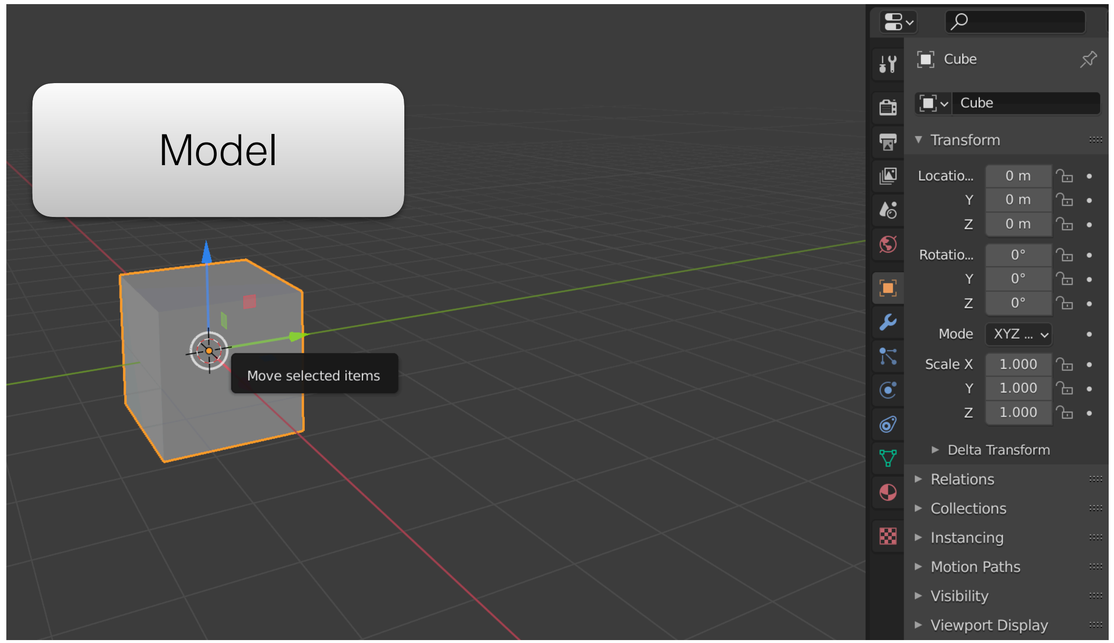


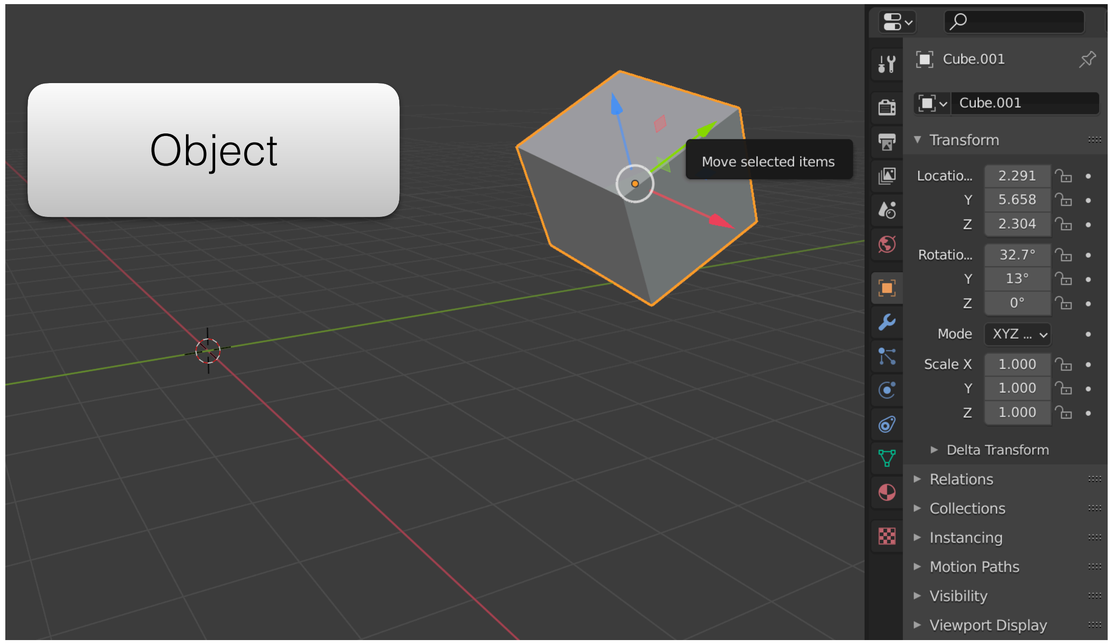
Geometry Processing - Vertex Shading



- Major tasks to be performed:
  1. compute the position for a vertex
  2. specify the outputs per vertex (normals, texture coordinates, …)
- The “vertex shader” name comes from the fact that, in the past, it was common to compute the final color for each vertex and have those colors interpolated across a triangle.
- More powerful hardware allowed the migration of the final color computation to be performed on a pixel level…
- … the vertex shader is now a general unit dedicated to setting up the data for each vertex (interpolation, animation, …)

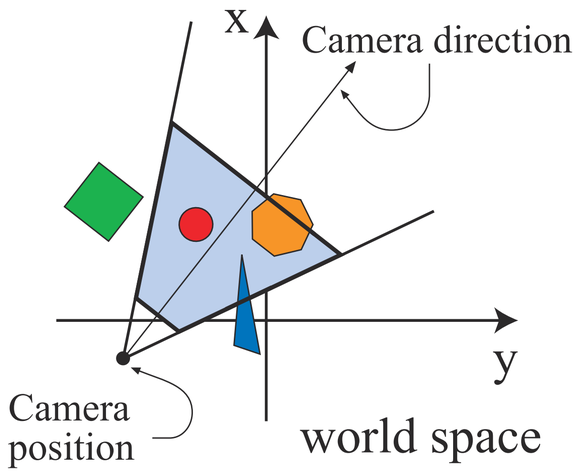
1. Compute the position of a vertex





- Go from models to objects in the scene by applying a modeling transformation to both vertices and normals
- The same model (base geometry) can be transformed using different modeling transformations (one for each instance)
- Original coordinates are defined in local coordinate system, while output coordinates are in a global/common coordinate system (World Coordinates)

1. Compute the position of a vertex

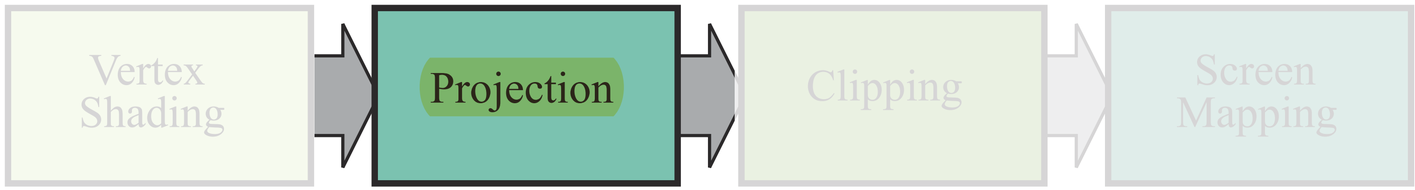


- Only objects captured by the camera need to be rendered
- The camera has a location and orientation in the world and it can be used to transform the vertices of the instances from world to camera coordinates (view transform)
- The local camera coordinates make projection and visible surface determination easier
- In camera coordinates, the camera is at the origin, looking towards the negative* z axis, with y pointing upwards
- Both the **model** transform and the **view** transforms can be implemented using 4x4 matrices and even be combined in a single matrix

2. Specify the outputs per vertex

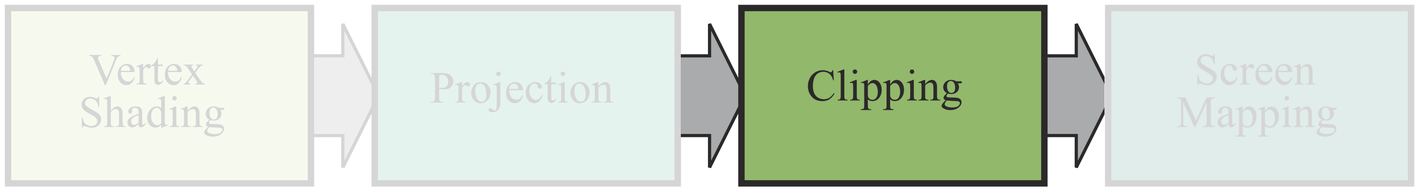
- To generate a realistic scene we need more than geometry…
- The appearance of the objects must be modeled using materials and light sources
- Determining the effect of light on a surface is called shading and it envolves evaluating a shading equation at several points of an object.
- From a set of per vertex input data (such as location, normals, texture coordinates, material properties or color) vertex shading results (which can be colors, vectors, texture coordinates, …) are sent to the rasterization and pixel processing stages to be interpolated and used in the final shading of the surface.

Geometry Processing - Projection



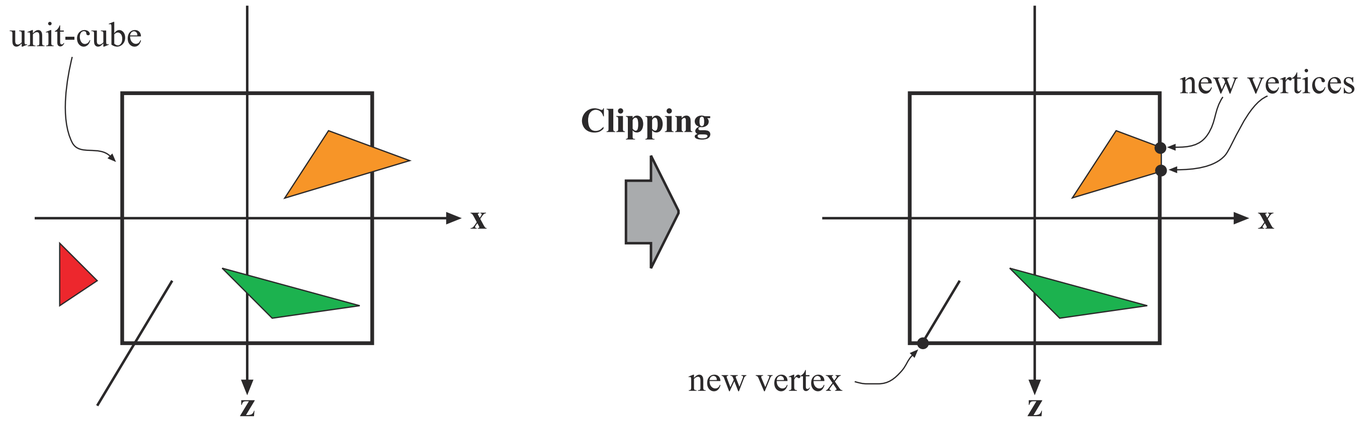
- Projection is the transformation of the view volume into a canonical view volume (typically a cube ranging from (-1,-1,-1) to (1,1,1)
- Both parallel and perspective view volumes (rectangular box and a truncated pyramid with rectangular base, respectively) can be transformed to this canonical view volume, simplifying the operations performed next
- Projection can also be expressed as a 4x4 matrix and concatenated with the previous transformations
- Although these transformations change a volume into another, they are called projections because after displaying the image, the z coordinate is stored in a different place (z-buffer) and the image generated is thus 2D only

Geometry Processing - Clipping



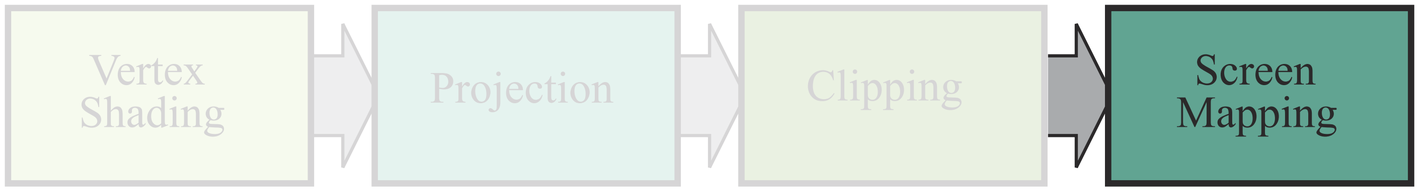
- After the projection transformation the objects are said to be in clip coordinates
- The GPU vertex shader must always output the corresponding clip coordinates of the input vertex
- Clipping is the process of throwing away the parts of a primitive that do not lie inside the canonical view volume.

Geometry Processing - Clipping



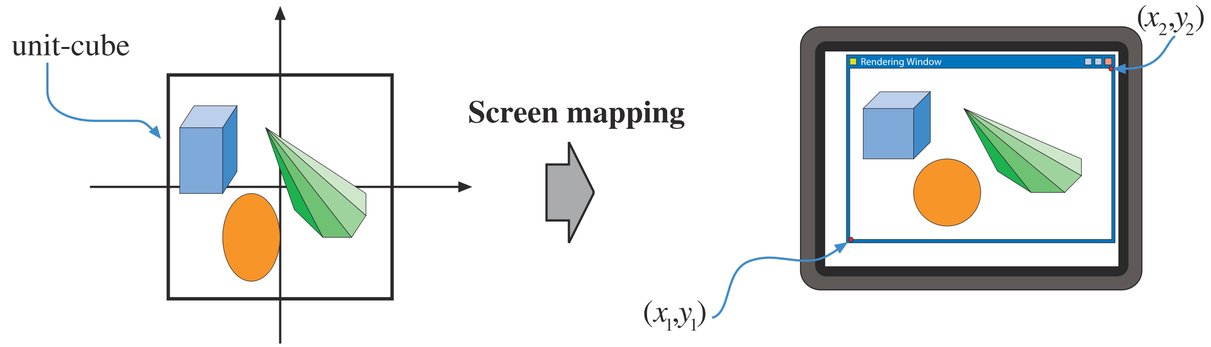
- After the projection transformation the objects are said to be in clip coordinates
- The GPU vertex shader must always output the corresponding clip coordinates of the input vertex
- Clipping is the process of throwing away the parts of a primitive that do not lie inside the canonical view volume
- Clipping is performed in homogeneous coordinates

Geometry Processing - Screen Mapping



- In the final stage, after clipping, the projected objects are transformed to screen space
- The rectangular area spanning from (-1,-1) to (1,1) is transformed to screen space using a viewport defined in integer screen coordinates.
- Screen coordinates together with the z coordinate are also called window coordinates.

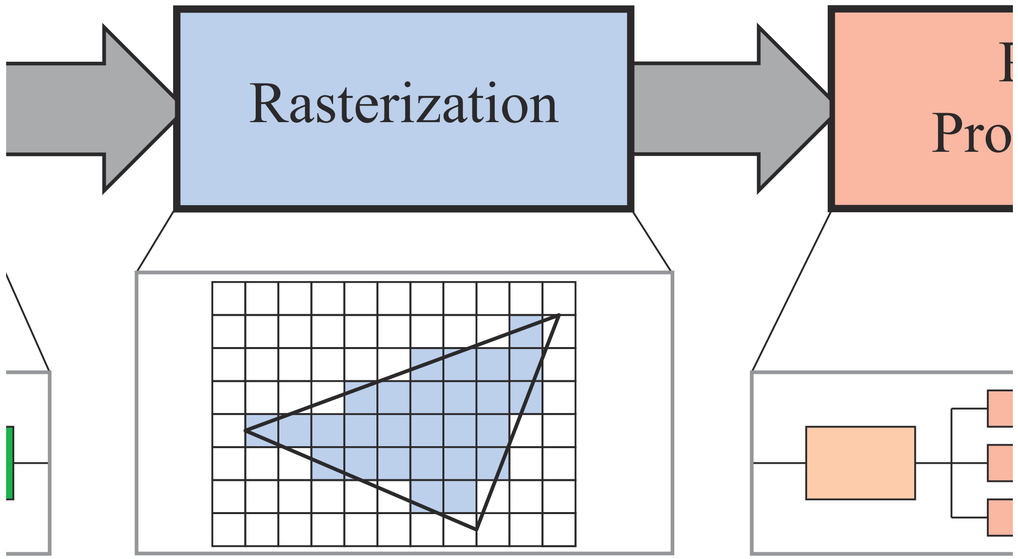
Geometry Processing - Screen Mapping



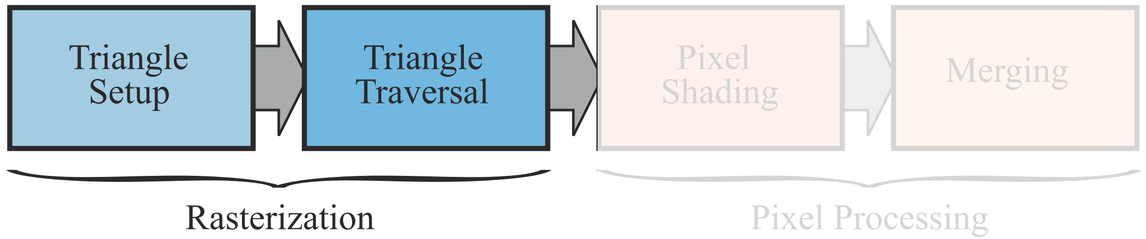
- The viewport spans from (x1, y1) to (x2, y2)
- Pixel centers have their locations at .5 (0.5, 1.5, 2.5, …)
- Left/bottom* side of pixel 0 has x/y coordinate equal to 0
- Left/bottom* side of pixel 1 has x/y coordinate equal to 1

Rasterization

- Takes as input a set of transformed and projected vertices and it finds the pixels that are considered as part of the primitive being drawn
- For a triangle, each set of three vertices result in a set of pixels considered to be inside the triangle (pixel centers inside the triangle) and that need to be further processed



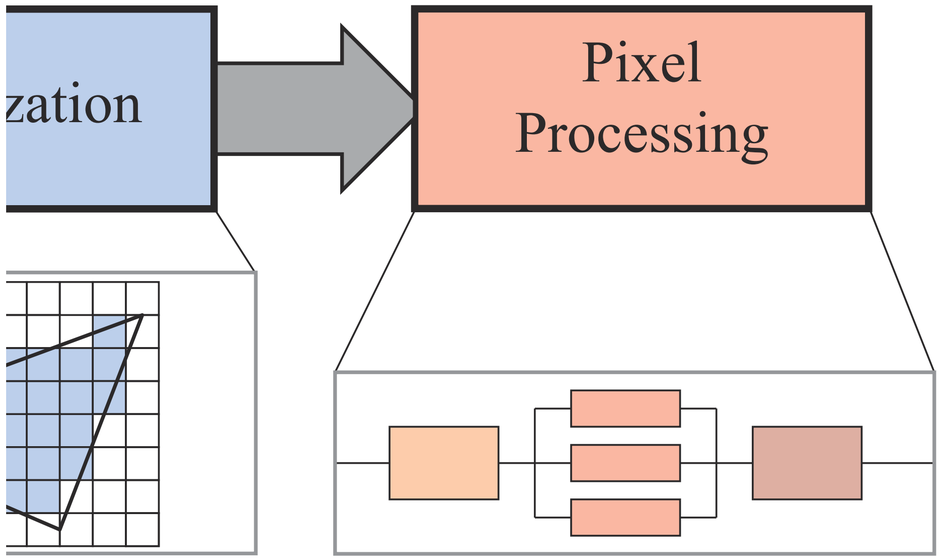
Rasterization



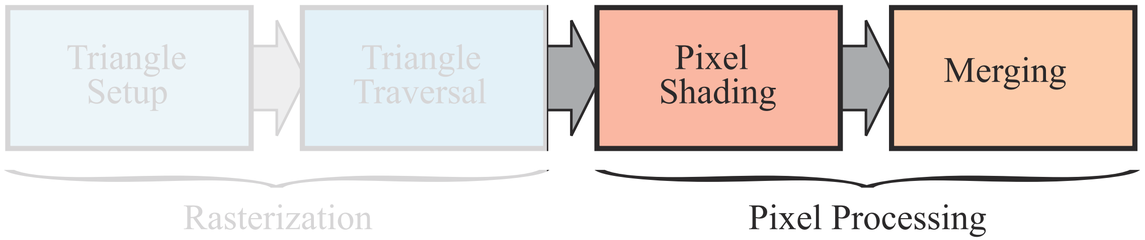
- Rasterization is subdivided in two functional stages
- The first stage is called Triangle Setup (or Primitive Assembly) and it picks groups of vertices to form the primitive (1 for points, 2 for lines and 3 for triangles). Triangles are born here! Differentials and edge equations are computed in this stage
- Triangle traversal generates a fragment for each pixel that has its center point covered by the primitive. Data associated with each fragment is interpolated from the data generated at each vertex of the primitive (includes depth and any shading data that was generated during vertex shading)
- All samples that are inside a primitive are then sent to the pixel processing stage

Pixel Processing

- In the final stage, each pixel is “shaded” by a program that computes its final color and may perform depth testing to determine if the pixel is visible
- Other per pixel operations such as blending the newly computed color with a previous color is also possible
- The rasterization and the pixel processing stages are totally performed in the GPU



Pixel Processing



- This stage is divided in two functional stages
- Pixel shading is performed on every sample using interpolated data generated before. This executes on a GPU and it is fully programmable. The pixel shader is provided by the programmer to shade each pixel and texture mapping is usually performed here.
- After a final color is computed for each fragment, it needs to be combined with whatever lies on screen at the output location. It is not a fully programmable stage but a highly configurable one. The visibility test for the fragment is performed in this stage (Z-Buffer)
- Besides color (frame buffer) and depth (z-buffer), we can also use the alpha channel (part of the frame buffer that handles transparency/opacity), the stencil buffer (a buffer where a primitive can also be written and that can be used to control rendering to the color buffer and the z-buffer)
- The operations at the end of the pipeline are raster operations or blend operations. The frame buffer is normally used to refer to all the buffers in the system. Double buffering (two color buffers) is used to avoid flickering or image tearing.

Further readings and resources

- Cap. 2 Real Time Rendering - T Akenine-Möller et. Al (adopted book)