Rasterization-based rendering

Rasterizaton-based rendering

- Rasterization-based rendering is quite old method extensively used and researched still today (and will stay with us for some time as we will see).
- Techniques used to produce an image with this method were developed 1960-1980*
- Rasterization is method used by GPUs to produce images. Rasterization-based techniques for image generation are so fundamental that they are deeply integrated within the GPU hardware architecture
- GPUs changed a lot from their beginnings but fundamental methods they implement for generating images haven't changed much. GPUs changed in efficiency and flexibility with some extensions. The pipeline and methods for generating images fundamentally stayed the same.

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

Rasterization: another solution to visibility problem

- When we introduced rendering, we divided it in two main tasks: visibility and shading.
- In raytracing-based rendering, we have seen that visibility can be divided into camera rays visibility (which objects are visible from camera) and secondary rays visibility (which surfaces are visible to each other during light transport in shading phase).
- Rasterization is another method for computing visibility. This method is extremely efficient for finding objects that are visible from camera and 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 visibiliy using rasterizer as it was in raytracing-based rendering, relies on geometrical techniques

Practical note: using rasterization-based renderer

- As discussed, rasterization-based rendering technique is deeply integrated in GPU hardware. Opposed to raytracing-based rendering which can be often found 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.
- In these lectures, we will have two passes on the topic:
  - One pass over concepts of rasterization-based rendering which should give the idea how can it be implemented.
  - One pass over GPU pipeline and graphical API

* 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/)

Concepts of rasterization 1

- 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.
  - 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-based rendering takes the opposite approach. It starts from objects (which are triangulated mesh) in 3D scene, projects triangles onto image plane and then computes which pixels of image plane are covered by the projected triangles.
-

Application of rasterization

- Rasterization is commonly employed in graphics rendering pipeline which is the term used in real-time rendering.
- Rasterization is only one part of graphics rendering pipeline (used for visibility calculation)
- Now we will discuss whole graphics rendering pipeline

Graphics rendering pipeline: a intro note

- For discussions on 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) is to render a 2D image from 3D scene (objects, lights, cameras)
- By now we have divieded rendering in two main stages:
  - Visibility calculation (which objects are visible from camera)
  - Shading calculation (light-matter calculation and light transport)
- Graphics rendering pipeline is performing exactly those but decomposed in smaller steps or stages.
  - 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 can be coarsely divided in four stages:
  - Application
  - Geometry processing
  - Rasterization
  - Pixel processing
- Each of the stages is usually pipeline itself: each consists of several substages.
- Note that there is difference between **functional stages –** task to be performed but not how and **implemented stages –** how are functional stages implemented in hardware and exposed to the user as API.

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 (thus executed on CPU*)
  - Efficiency of this stage is propagated to further stages: e.g., sending less geometry on GPU
- 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)
  - others depending on application which subsequent stages of pipeline can not handle.
- Geometry to be rendered is sent to geometry processing stage – these are called rendering primitives (e.g., points, lines and triangles) which might end up on display device

*Some application work can be send to GPU for processing but not as next stage. This kind of mode is called **compute shader –** treats GPU as highly parallel general processor and not as rendering function.

Geometry processing stage

- Typically performed on GPU
- Responsible for most of the per-triangle and per-vertex operations
  - Deals with transformations, projections and all other geometry handling
- This stage computes what is drawn, how and where
- It is further divided into following functional stages:
  - Vertex shading
  - Projection
  - Clipping
  - Screen mapping

Vertex shading*

- Two main tasks:
  - Compute position of vertex
  - Evaluate/set-up any additional vertex data output desired by programmer such as normal and texture coordinates
- An example of vertex shader task is animating objects using transformations on vertices.
  - Vertex blending – RTR 4.4
  - Morphing – RTR 4.5.

* Traditionally, shading was performed by applying light at vertex position and normal. Result was stored as color per vertex and interpolated across triangle. This can be also used today if 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.

Vertex shading: 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
  - World
  - View/camera
  - Clip

<IMAGE: BIG PICTURE OF TRANSFORMATIONS>

Model space

- Originally, model that is given to vertex shading stage is in its own space – **model space**. This means that model has not been transformed at all.
- Each model can be associated with **model transform –** for positioning and orienting
  - 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.
  - <IMAGE: INSTANCING>
- Model transform is applied on model's vertices and normals. Coordinates of a object are also called model **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 is unique and after all models have been transformed with their respective model transforms, they all exist in the same space – world space.

<EXAMPLE: world space is actually scene. Moving models to world space with their model transforms is actually part of modeling>

Camera (view, eye) space

- Not all objects in the world space are visible from camera and thus not rendered.
- Camera has a location and orientation in world space which can be used for positioning and aiming the camera.
- For further steps of the pipeline: projection and clipping, camera and all objects are tranformed with **view (camera) transform**.
- With view transform, camera is placed in world origin and aimed in negative z axis with y pointing up and x pointing right*.
- After transform, the model is said to be in **camera (view or eye) space.**

<EXAMPLE: MODEL TO VIEW SPACE>

- Practical note: 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.

* 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.

Vertex shading: additional data

- By now we discussed how vertices are processed in vertex shading stage. This data gives information about object shape but not its appearance.
- Appearance is dependent on object **material and light sources** in the scene.
  - Materials and lights can be modeled in any number of ways: from simple color to elaborated physical descriptions which determine or are used for computing the color.
- Alongside vertex position and normal, material and light sources data are used in **shading.** Shading involves computing the shading equation which relies on those data.
- **Shading** is performed at various points on object. It **can be** performed both in vertex shading stage or pixel processing stage.
- Data used for shading can be stored per vertex: location, normal, material parameters (e.g., color, texture coordinates) or any other numerical information needed for evaluating shading equation.
- All the mentioned data is sent to rasterization and pixel processing stage for interpolation across triangle and shading in each pixel of triangle.

Vertex shading: projection

- After models are transformed to camera space, a projection and clipping is applied.
- Projection and clipping transforms the view volume in a unit cube with points (-1,-1,-1) and (1,1,1) – a **canonical view volume**.
- Projection is done first. Two commonly used projection methods:
  - Orthographic – one type of parallel projections
  - Perspective
- Projection is represented as matrix and can be combined with other geometry transforms: model and camera.
- <PERSPECTIVE MATRIX>
- <IMAGE: PERSPECTIVE VS ORTHOGRAPHIC>
  - View volume of orthographic projection is rectangular box – parallel lines remain parallel. Combination of translation and scaling. 4X4 matrix
  - IN perspective projection, further objects appear smaller. Parallel lines may converge at the horizon. The view volume is called fustum – truncated pyramid with rectangular base. Frustum is transformed in unit cube as well. 4X4 matrix
- After projection, z-coordinate is not stored in generated image but as a **z-buffer –** this ways model is projected from three to two dimensions.

Vertex shading: clip coordinates

- After perspective transforms have been applied on model, the model coordinates are said to be clip coordinates.
- These are homogeneous coordiantes (before division with w)
- Vertex stage must always output coordinates in this type for next functional stage: clipping to work correctly.

Optional vertex processing

- Each rendering pipeline has vertex processing as described
- 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.

Geometry processing: 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 view and projection transformations, clipping of primitive is always done against unit cube.
  - <EXAMPLE: CLIPPING PROCESS FOR TRIANGLE>
    - Vertex that is outside of view volume is removed. New vertices are created on clipping position.
- Additional clipping planes can be introduced by programmer to chop the visibility of objects
- Clipping uses 4 homogeneous coordinates produced by projection. The 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).

Screen mapping

- Clipped primitives inside view volume are passed on **screen mapping**.
- Coordinates entering this stage are still 3D.
- X 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.

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*:
  - Triangle setup
  - Triangle traversal

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

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.

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).
  - Properties: fragment depth and any shading data from geometry processing stage.
- All pixels, that is fragments are sent to pixel processing stage.

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:
  - Pixel shading
  - 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.

Pixel processing: pixel shading

Pixel processing: pixel merging

Review of the pipeline

- RTR 2.6

Application of graphics rendering pipeine

- 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