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

- 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 coarsly 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
- Deals with transformations, projections and all other geometry handling
- This stage computes what, how and where it is drawn

Rasteriuzation stage

- Typically takes input of 3 vertices forming a triangle, finds all pixels that are considered inside the triangle and forwards those further
- Runs on GPU.

Pixel processing stage

- 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.

Application of graphics rendering pipeine

- Graphics rendering pipeline is implemented on GPUs and is basis for real-time rendering

Graphics rendering pipeline: OpenGL