libgraphics: Design and Implementation

Rodrigo G. López
rgl@antares-labs.eu

Introduction

Libgraphics is a 3D computer graphics library that provides a way to set up a scene, fill it up with a bunch of models (with their own meshes and materials), lights and cameras, and start taking pictures at the user request. It implements a fully concurrent retained mode software renderer, with support for vertex and fragment/pixel shaders written in C (not GPU ones, at least for now), a z-buffer, front- and back-face culling, textures and skyboxes, directional and punctual lights, tangent-space normal mapping, ???

The renderer

The *renderer* is the core of the library. It follows a **retained mode** model, which means that the user won't get a picture until the entire scene has been rendered. Thanks to this we can also clear and swap the framebuffers without their intervention, they only need to concern themselves with shooting and "developing" a camera.

It's implemented as a tree of concurrent processes connected by `Channels`—as seen in **Figure 1**—, spawned with a call to `initgraphics`, each representing a stage of the pipeline:

The **renderer** process, the root of the tree, waits on a `channel` for a `Renderjob` sent by another user process, specifying a scene, a camera and a shader table. It walks the scene and sends each `Entity` individually to the entityproc.

The **entityproc** receives an entity and splits its geometry equitatively among the tilers, sending a batch for each of them to process.

Next, each **tiler** gets to work on their subset of the geometry (potentially in parallel)—see **Figure 2**. They walk the list of primitives, then for each of them apply the **vertex shader** to its vertices (which expects clip space coordinates in return), perform frustum culling and clipping, back-face culling, and then project them into the viewport (screen space). Following this step, they build a bounding box, used to allocate each primitive into a rasterization bucket, or **tile**, managed by one of the rasterizers; this is illustrated in **Figure 3**. If it spans multiple tiles, it will be copied and sent to each of them.

Finally, the **rasterizers** receive the primitive in screen space, slice it to fit their tile, and apply a rasterization routine based on its type (only *points*, *lines* and *triangles* are supported). For each of the pixels, a **depth test** is performed, discarding fragments that are further away. Then a **fragment shader** is applied and the result written to the framebuffer.

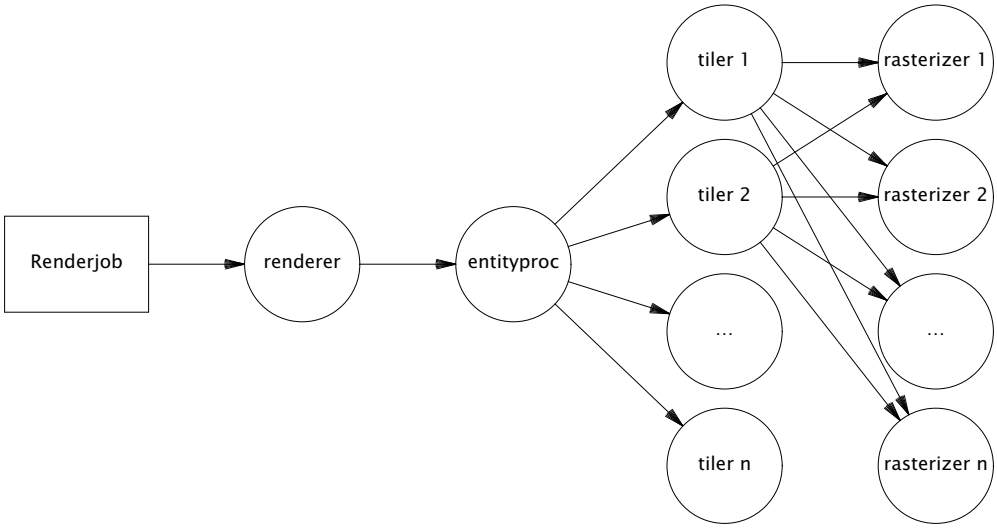


Figure 1: The rendering graph for a **2n** processor machine.

Tile-based rendering

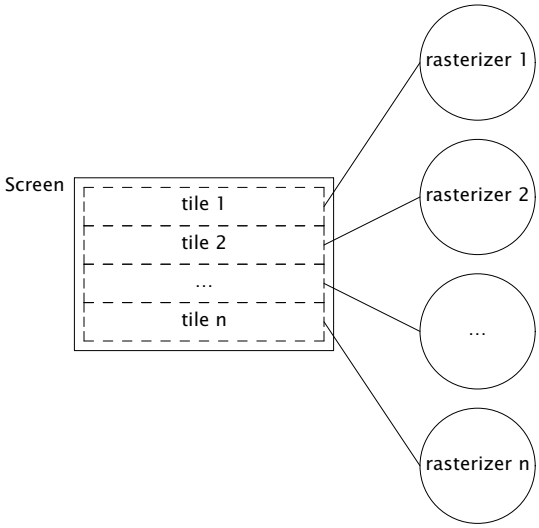


Figure 2: Per tile rasterizers.

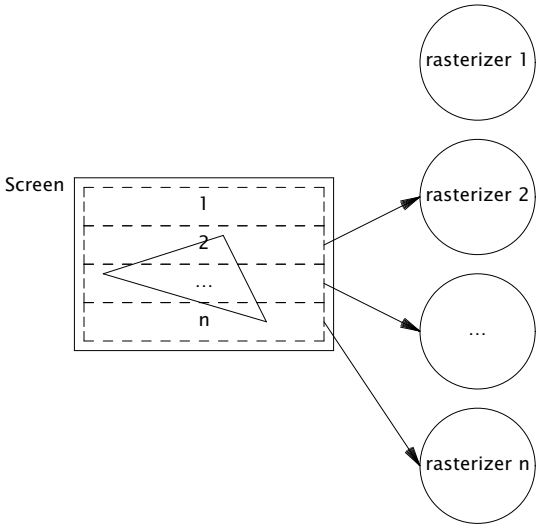
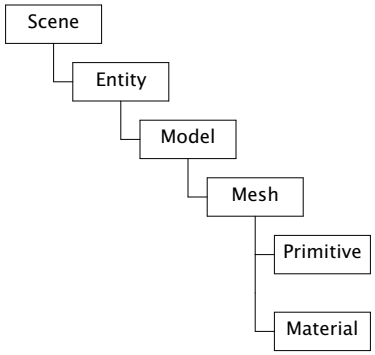


Figure 3: Raster task scheduling.

The scene



Frames of reference

Frames are right-handed throughout every stage.

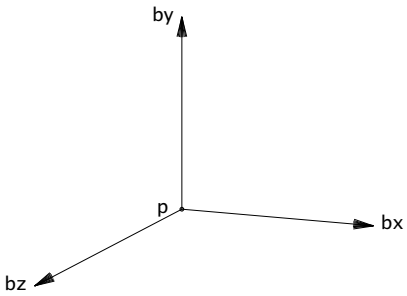


Figure 4: Example right-handed rframe.

Viewports

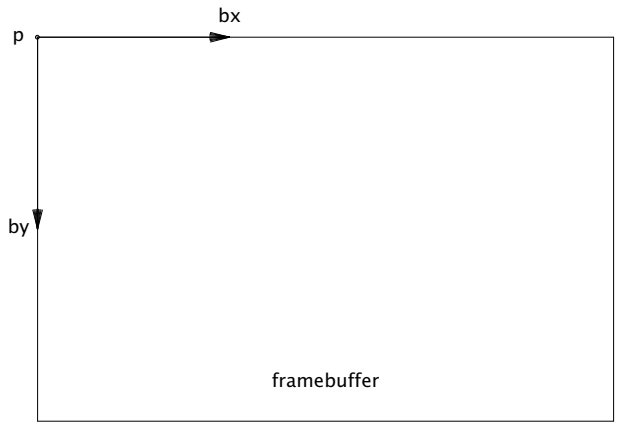


Figure 5: Illustration of a 3:2 viewport.