libgraphics: Design and Implementation

Rodrigo G. López
rgl@antares−labs.eu

ABSTRACT

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), and featuring a z-buffer, front- and back-face culling, textures and skyboxes, directional and punctual lights, tangent-space normal mapping, among other things.

Introduction

Write the intro last.

1. The scene

```
struct Scene
{
	char *name;
	Entity ents;
	ulong nents;
	Cubemap *skybox;

	void (*addent)(Scene*, Entity*);
	void (*delent)(Scene*, Entity*);
};
```

A *scene* is a container, represented as a graph, that hosts the entities that make up the world. Each of these entities has a model made out of a series of meshes, which in turn are made out of geometric primitives (only *points*, *lines* and *triangles* are supported). Each model also stores a list of materials.

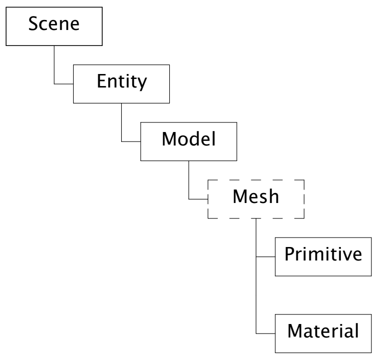

Figure 1: The scene graph.

1.1. Entities

```
struct Entity
{
        RFrame3;
        char *name;
        Model *mdl;

        Entity *prev, *next;
};
```

Entities represent physical objects in the scene.

1.2. Models

```
struct Model
{
        Primitive *prims;
        ulong nprims;
        Material *materials;
        ulong nmaterials;
};
```

1.3. Meshes

1.4. Primitives

```
struct Primitive
{
        int type;
        Vertex v[3];
        Material *mtl;
        Point3 tangent; /* used for normal mapping */
};
```

1.5. Materials

```
struct Material
{
        char *name;
        Color ambient;
        Color diffuse;
        Color specular;
        double shininess;
        Texture *diffusemap;
        Texture *normalmap;
};
```

2. Cameras

3. 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 2**—, spawned with a call to `initgraphics`, each representing a stage of the pipeline:

3.1. renderer

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

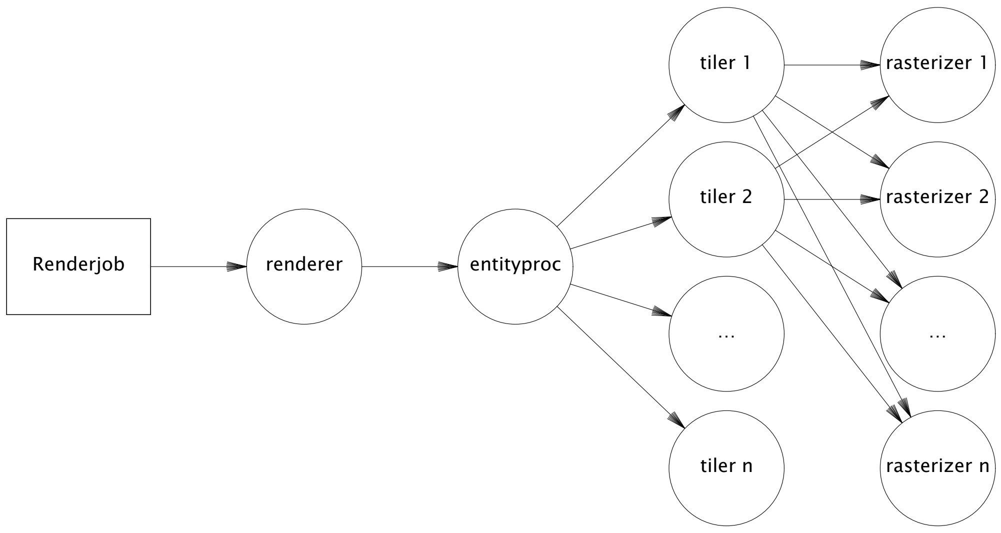

Figure 2: The rendering graph for a **2n** processor machine.

3.2. entityproc

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

3.3. tilers

Next, each **tiler** gets to work on their subset of the geometry, potentially in parallel—see **Figure 3**. 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 to obtain their screen space coordinates. 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; as illustrated in **Figure 4**. If it spans multiple tiles, it will be copied and sent to each of them.

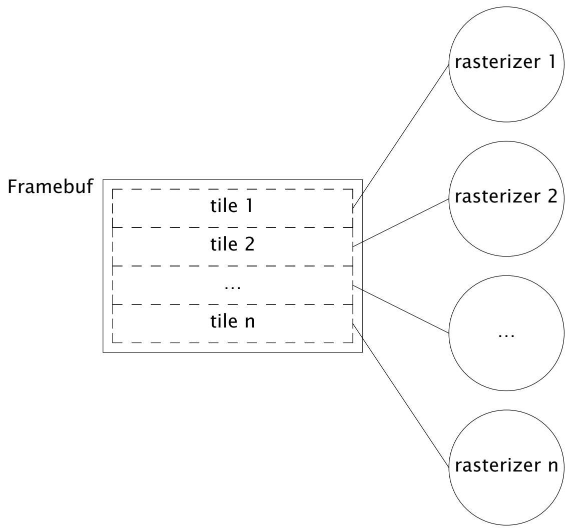

Figure 3: Per tile rasterizers.

3.4. rasterizers

Finally, the **rasterizers** receive the primitive in screen space, slice it to fit their tile, and apply a rasterization routine based on its type. 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 after blending.

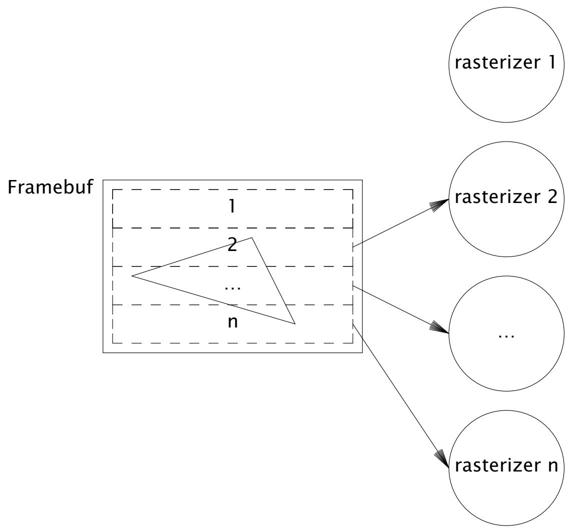

Figure 4: Raster task scheduling.

4. Frames of reference

Frames are right-handed throughout every stage.

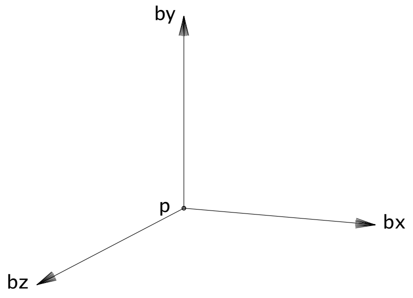

Figure 5: Example right-handed rframe.

5. Viewports

A *viewport* is a sort of virtual framebuffer, a device that lets users configure the way they visualize a framebuffer, which changes the resulting *image*(6) after a call to its `draw` or `memdraw` methods. So far the only feature available is upscaling, which includes user-defined filters for specific ratios, such as the family of pixel art filters *Scale[234]x*, used for 2x2, 3x3 and 4x4 scaling respectively*[REF01]*. Users control it with calls to the viewport's `setscale` and `setscalefilter` methods.

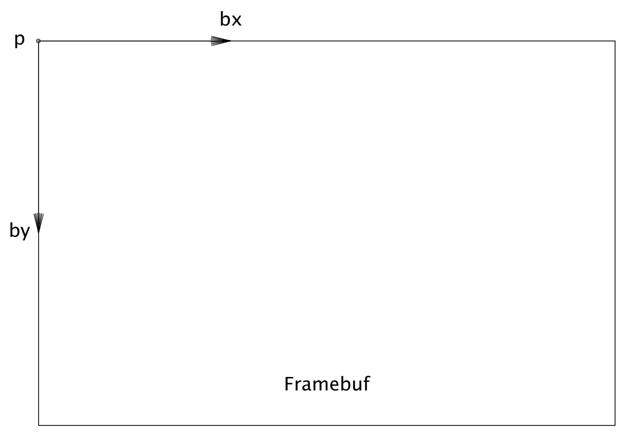

Figure 6: Illustration of a 3:2 viewport.

References

[REF01]https://www.scale2x.it/