libgraphics: Design and Implementation

Rodrigo G. López
rgl@antares-labs.eu

ABSTRACT

Libgraphics is a 3D computer graphics library for Plan 9. It implements a fully concurrent retained mode software renderer for polygon rasterization, and supports vertex and pixel shaders written in C (not GPU ones, at least for now[1]), a z-buffer, front- and back-face culling, textures, skyboxes, and directional and punctual lights, among other things.

Introduction

Write the intro last.

1. The scene

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

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

A *scene* is a container, represented as a graph, that hosts the entities that make up the world, as well as properties of it such as lighting and the skybox.

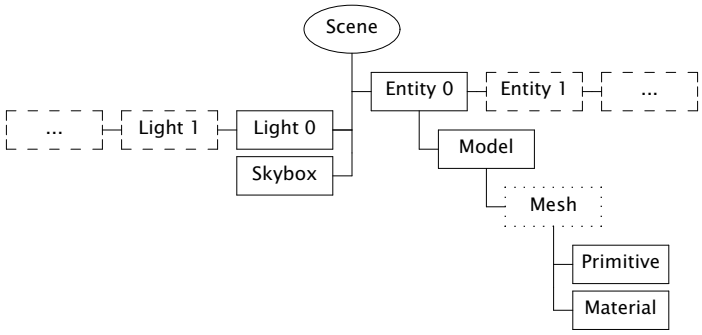

Figure 1: The scene graph.

1.1. Entities

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

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

Entities represent visible physical objects in the scene. Each of these entities has a frame of reference to locate and orient it in the world, a unique name and a localized model that encodes its physical properties.

1.2. Models

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

        int (*addprim)(Model*, Primitive);
        int (*addmaterial)(Model*, Material);
};
```

A model contains the geometric and material properties of an *Entity*. The geometry is encoded as a list of primitives, which in turn refer to any out of a list of materials that encode the visual parameters that determine its appearance.

1.3. Meshes

Meshes are not implemented yet, but the idea is for them to provide a hierarchy of primitives with which to apply optimizations (for storage, visibility determination, maybe others.)

1.4. Primitives

```
struct Primitive
{
        int type;
        Vertex v[3];
        Material *mtl;
        Point3 tangent;
};
```

Primitives are geometric building blocks, namely points, lines and triangles. The tangent is used in triangles for warp-safe normal mapping.

1.5. Materials

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

A material defines the optical characteristics of a surface.

2. Cameras

```
struct Camera
{
        RFrame3;                /* VCS */
        Viewport *view;
        Scene *scene;
        Renderer *rctl;
        double fov;             /* vertical FOV */
        struct {
                double n, f;    /* near and far clipping planes */
        } clip;
        Matrix3 proj;           /* VCS to clip space xform */
        Projection projtype;
        int cullmode;
        uint rendopts;
};
```

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 apply optimizations to make better use of the pipeline, clear and swap the framebuffers, and—in the future—run distributed rendering jobs, all without any intervention; users only need to concern themselves with shooting and "developing" a camera.

It's implemented as a tree of concurrent processes connected by buffered `Channels`—as seen in **Figure 2**—, spawned with a call to `initgraphics`, each representing a stage of the pipeline:

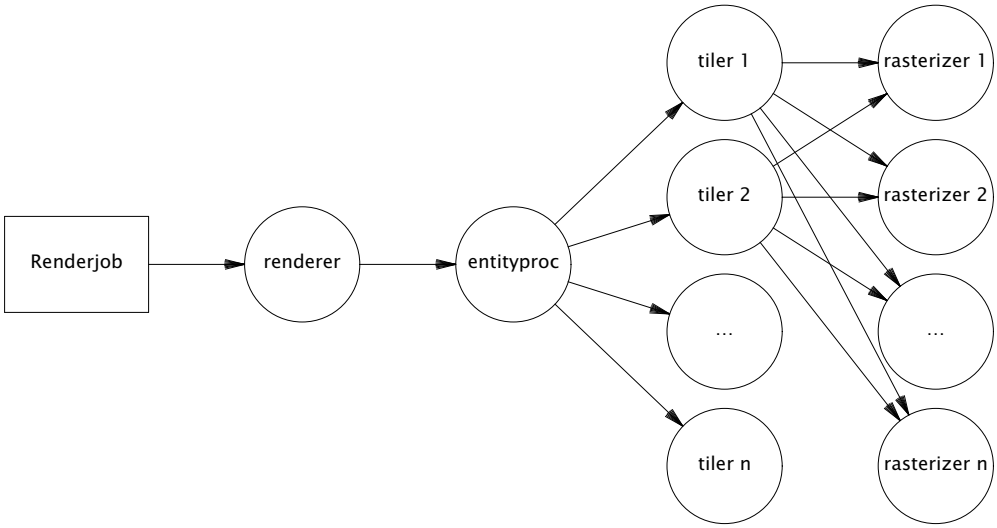

Figure 2: The rendering graph for a **2n** processor machine.

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**.

3.2. entityproc

The **entityproc** receives an entity and splits its geometry equitatively among the **tiler**s, 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 **rasterizer**s; as illustrated in **Figure 4**. If it spans multiple tiles, it will be copied and sent to each of them.

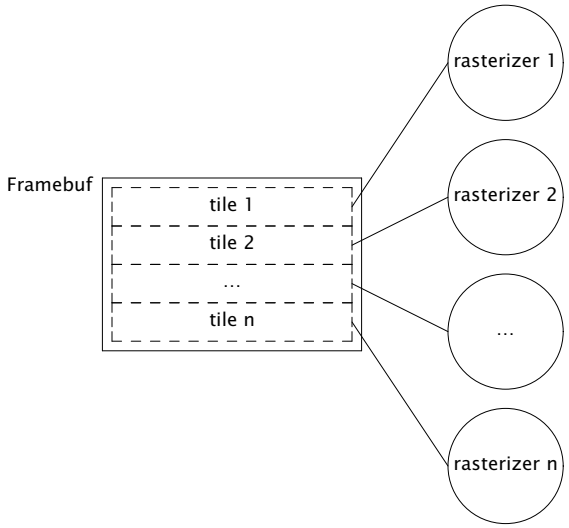

Figure 3: Per tile rasterizers.

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.

> Depth testing and blending can be disabled by clearing the camera's `RODepth` and `ROBlend` bits from the `rendopts` property, respectively. An experimental A-buffer implementation is also included for order-independent rendering of transparent primitives (OIT). If enabled, by setting the camera's `ROAbuff` bit, fragments will be pushed to a depth-sorted stack, waiting to be blended back-to-front and written to the framebuffer at the end of the job.

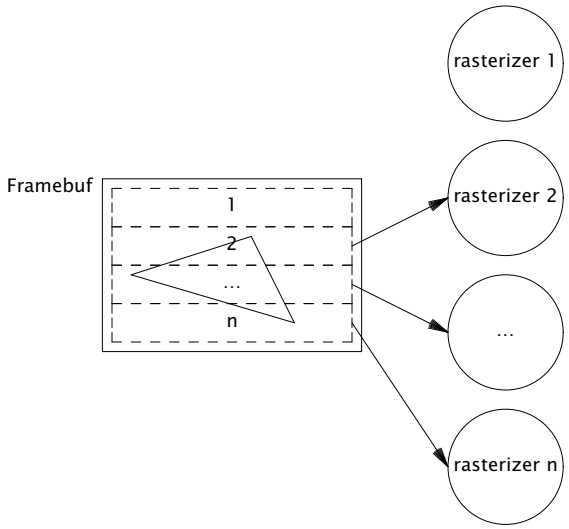

Figure 4: Raster task scheduling.

4. Frames of reference

Frames are right-handed throughout every stage of the pipeline, as well as in the world. A camera that's looking at an object has its Z-axis basis (`bz`) pointing away from it.

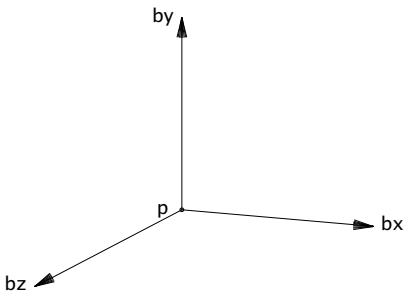

Figure 5: Example right-handed rframe.

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*[2]*. Users control it with calls to the viewport's `setscale` and `setscalefilter` methods.

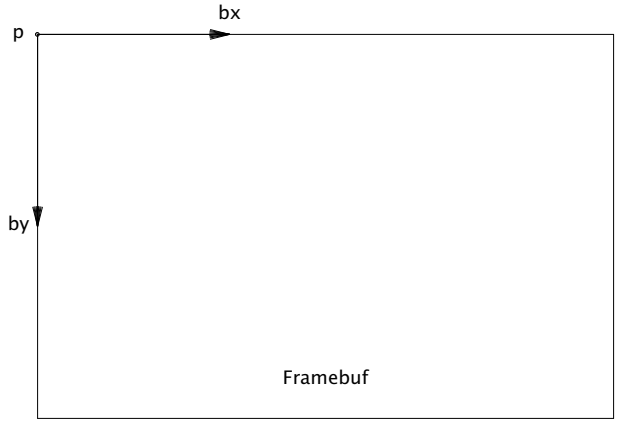

Figure 6: Illustration of a 3:2 viewport.

References

[1] https://shithub.us/sirjofri/gpufs/HEAD/info.html

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

[3] Thomas W. Crockett, "Design Considerations for Parallel Graphics Libraries", *NASA Langley Research Center, Contract Nos. NAS1-18605 and NAS1-19480, June 1994*

[4] Thomas W. Crockett, "Parallel Rendering", *NASA Langley Research Center, Contract No. NAS1-19480, April 1995*

[5] Thomas W. Crockett, "Beyond the Renderer: Software Architecture for Parallel Graphics and Visualization", *NASA Langley Research Center, Contract No. NAS1-19480, December 1996*

[6] Tomas Akenine-Möller et al, "Real-Time Rendering", *4th edition, Taylor & Francis,*

CRC Press, 2018

[7] James F. Blinn, Martin E. Newell, “Clipping Using Homogeneous Coordinates”, *SIGGRAPH ’78: Proceedings, August 1978, pp. 245–251*

[8] “GPU Gems” series

[9] “Graphics Gems” series

[10] Ian Stephenson, “Production Rendering: Design and Implementation”, *Springer, 2005*

[11] Paul S. Heckbert, “Survey of Texture Mapping”, *IEEE Computer Graphics and Applications, Nov. 1986, pp. 56–67*

[12] Paul S. Heckbert, “Fundamentals of Texture Mapping and Image Warping”, *University of California, Berkeley, Technical Report No. UCB/CSD–89–516, June 1989*

[13] Robert L. Cook, Loren Carpenter, Edwin Catmull “The REYES Image Rendering Architecture”, *ACM Transactions on Computer Graphics, Vol. 21, No. 4, July 1987*

[14] Bruce J. Lindbloom, “Accurate Color Reproduction for Computer Graphics Applications”, *ACM Transactions on Computer Graphics, Vol. 23, No. 3, July 1989*