Beyond Triangles: Using Ray-Casting to Render Non-Triangle Primitives Directly In Graphics Hardware

Walter Mundt, Sumanta Pattanaik, Erik Reinhard
University of Central Florida

We introduce a simple, efficient framework for integrating arbitrary ray-casted primitives into scenes rendered with more conventional techniques. This framework will allow for the rapid introduction of increasingly complex primitive types as graphics hardware and rendering technology advance.

The overall organization of the framework is shown in Figure 1. The framework defines an object-oriented interface to the *primitive renderers*, a common vertex shader, and an HLSL function to perform lighting. Each primitive renderer is responsible for providing a set of C++ functions for managing primitive-specific GPU resources, as well as a *driver* function that performs the actual rendering. The framework also allows for a primitive renderer to implement an optional interface to allow for GPU-based object choosing and manipulation.

The most important part of each primitive renderer is a shader that performs per-pixel geometry processing. Most of these use ray-casting to determine the depth, and normal of the primitive point along the ray through current pixel. The renderer's driver code provide the information required for these calculations. The pixel shader outputs depth information, and standard depth buffering handles occlusion (see Figures 2 and 3). Using the depth buffer allows seamless integration with the standard methods for rendering triangles (see example in Figure 4). Similarly, we use shadow mapping to provide shadow support that is independent of the nature of the primitives being rendered.

It's performance-critical to minimize the number of pixels for which ray-intersection tests must be performed. Every ray-casted primitive renderer provides its bounding box information. The vertex shader does the standard matrix transformation on the bounding box, and also calculates model-space ray directions at the vertices. These ray directions are interpolated by the hardware as texture coordinates, and are used by the renderers' pixel shaders to perform ray-intersection tests. Object definitions are either provided as a set of shader constants, or encoded within the vertex structure. No precomputation is generally required, so the framework lends itself very well to the highly-dynamic scenes often found in hardware rendering applications.

We compare the performance of our algorithms with that of a standard pipeline by rendering *sphereflake*, a fractally-generated set of spheres from the Standard Procedural Database (see Table 1). The images for comparison are rendered at a resolution of 600×600, with a 1024×1024 shadow map on a Radeon-9800 (see Figure 5). The standard-pipeline renderer uses a primitive LOD scheme where smaller spheres are tessellated into fewer polygons. This results in some visible artifacts as the viewpoint approaches the sphere-flake.

REFERENCES

CARR, N. A., HALL, J. D., AND HART, J. C. 2003. The ray engine. In *Proceedings of the EUROGRAPHICS conference on Graphics hardware*, 37–46.

PARKER, S., MARTIN, W., SLOAN, P.-P., SHIRLEY, P., SMITS, B., AND HANSEN, C. 1999. Interactive ray tracing. In *Symposium on Interactive 3D Computer Graphics*, 119–126.

PURCELL, T. J., BUCK, I., MARK,W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. In *Proceedings of the SIGGRAPH-02*, 703–712.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3, 153–164.

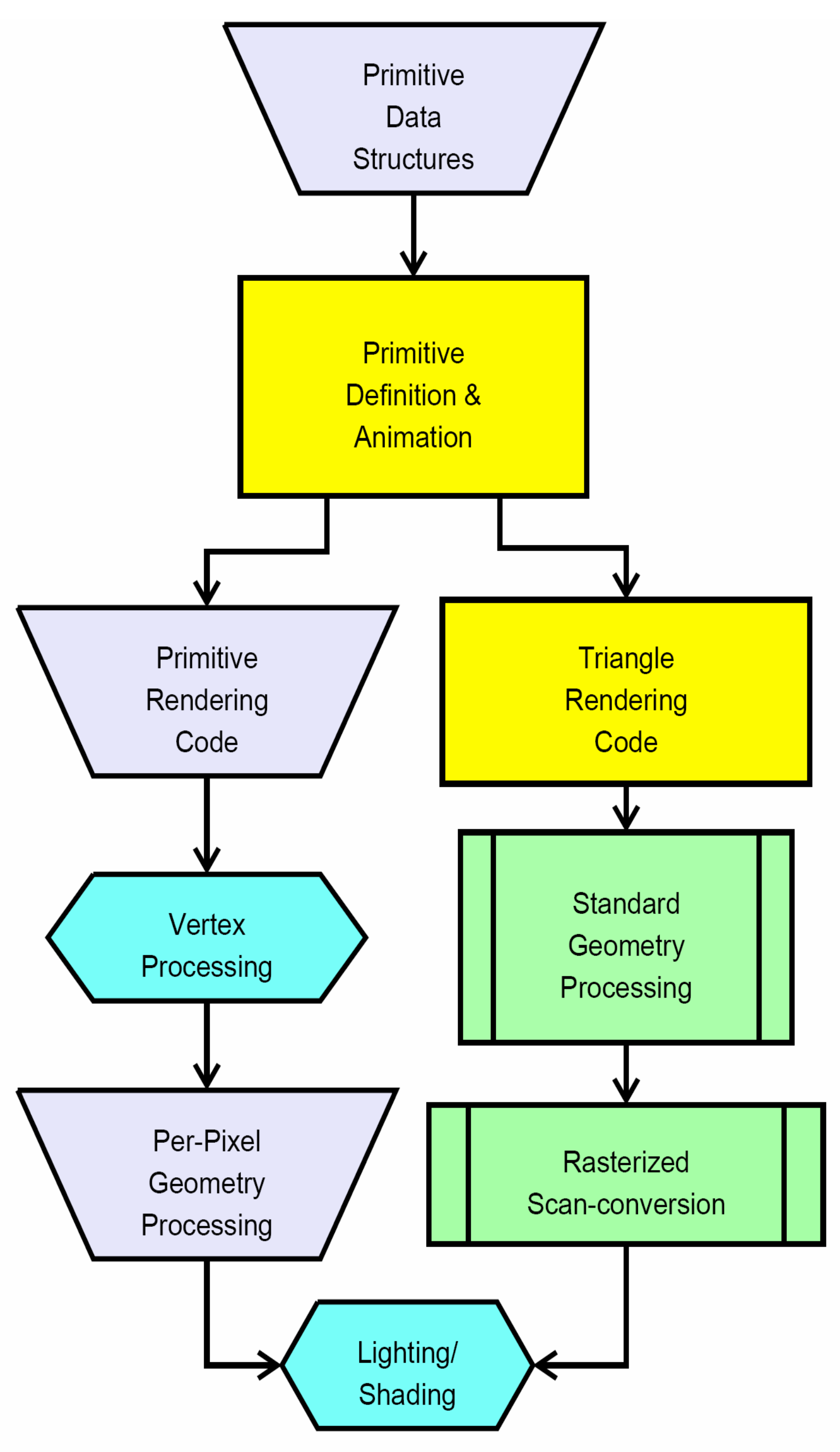

Primitive Data Structures
Primitive Definition & Animation
Primitive Rendering Code
Triangle Rendering Code
Vertex Processing
Standard Geometry Processing
Per-Pixel Geometry Processing
Rasterized Scan-conversion
Lighting/ Shading

Fig. 1. *Diagram of framework architecture*

ACKNOWLEDGEMENTS

This work was partially supported by ATI-Research, I-4 Matching fund and Office of Naval Research.

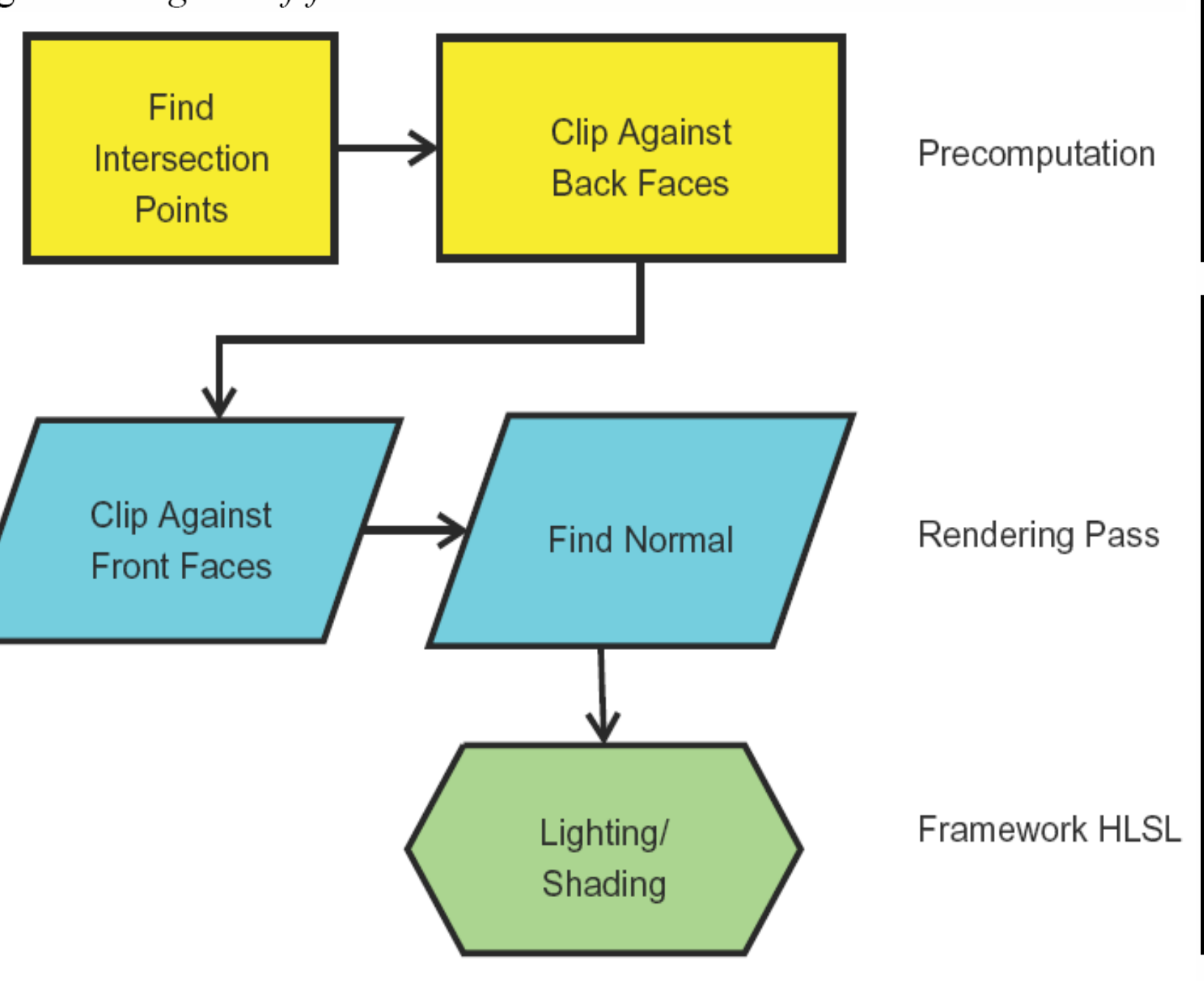


Fig. 2. *Quadric Renderer Overview*

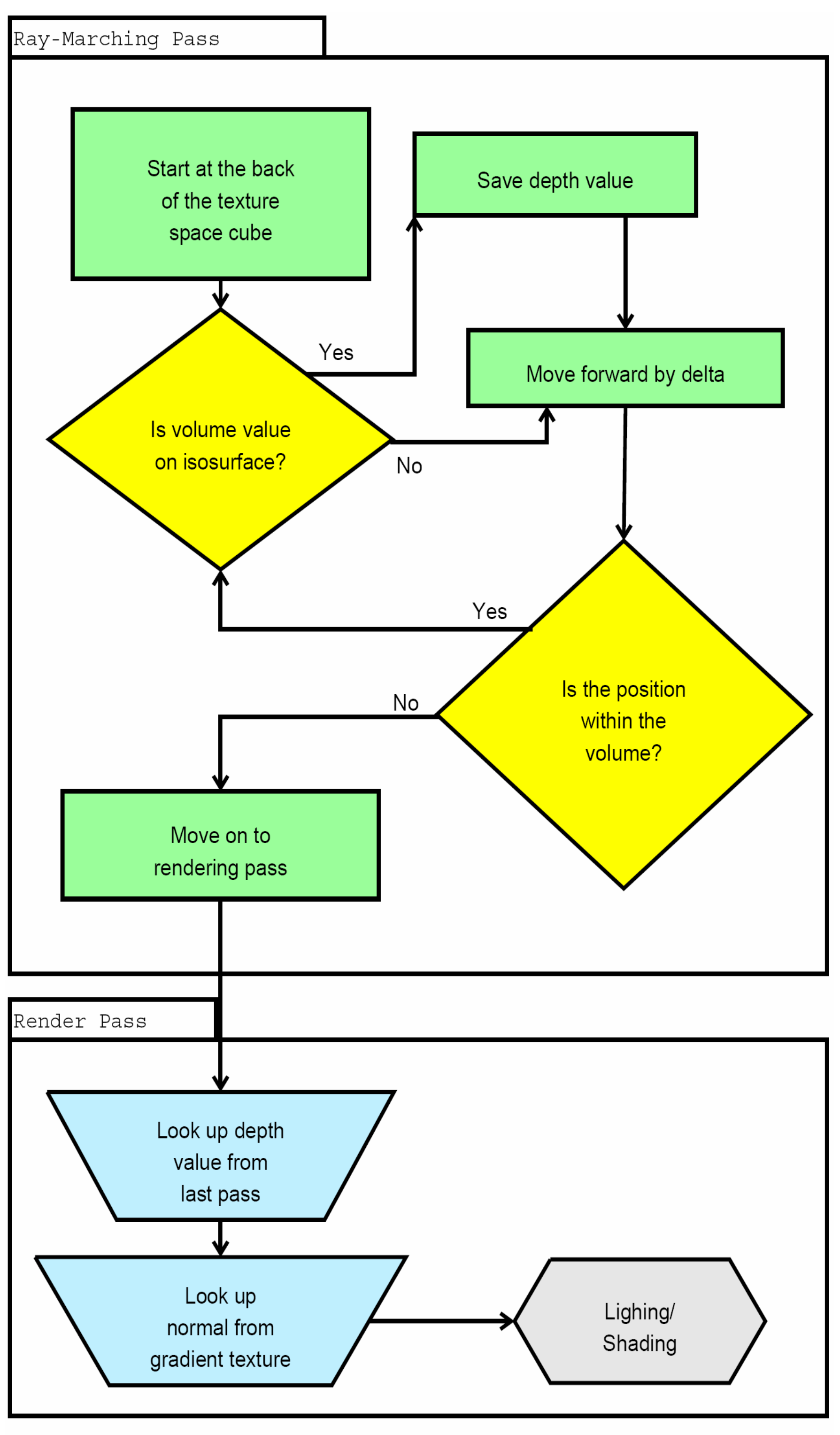
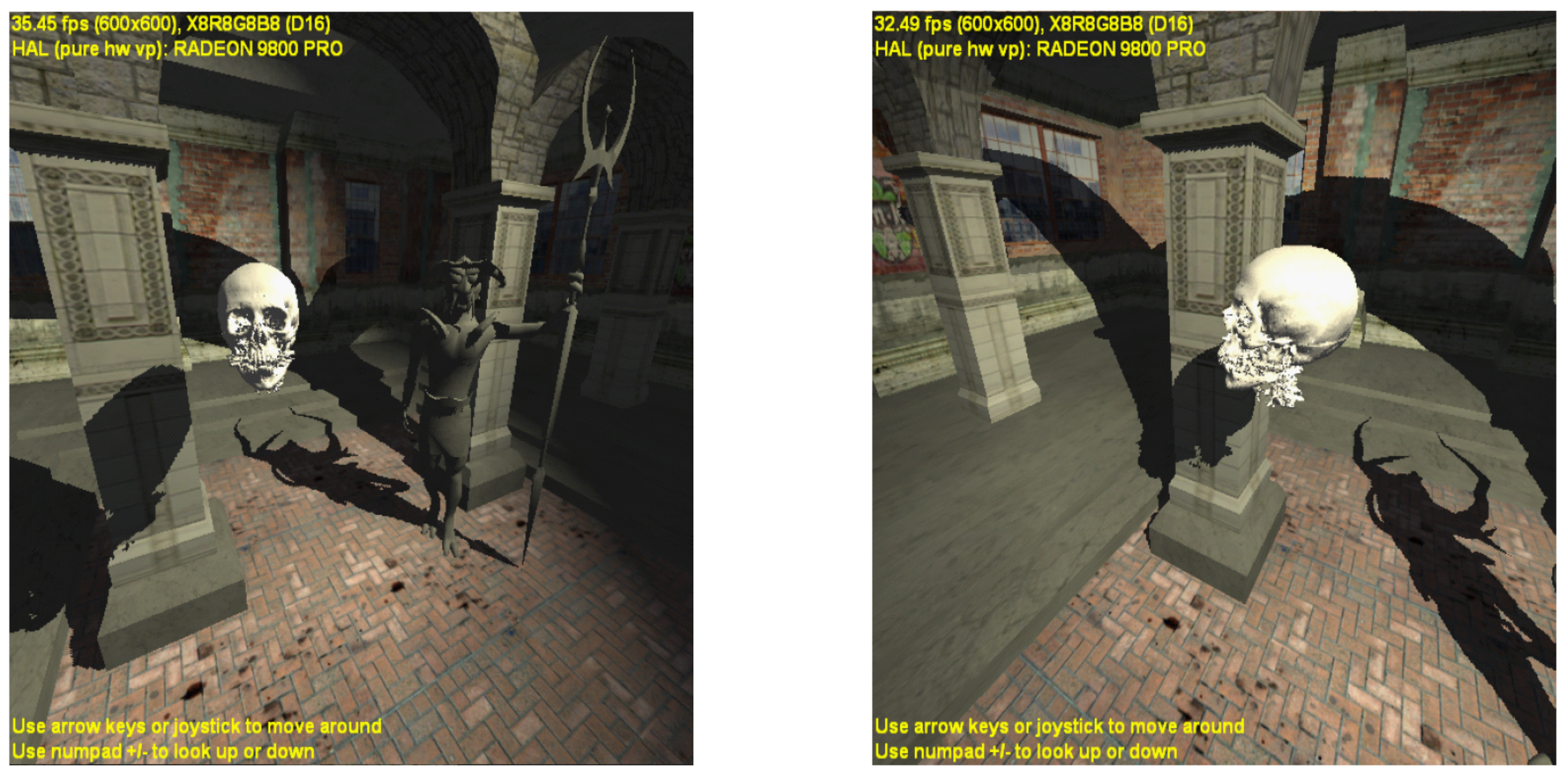

Ray-Marching Pass
Start at the back of the texture space cube
Save depth value
Yes
Move forward by delta
Is volume value on isosurface?
No
Yes
No
Is the position within the volume?
Move on to rendering pass
Render Pass
Look up depth value from last pass
Look up normal from gradient texture
Lighing/ Shading

Fig. 3. *Volume Rendering Flowchart*



35.45 fps (600x600), X8R8G8B8 (D16)
HAL (pure hw vp): RADEON 9800 PRO
Use arrow keys or joystick to move around
Use numpad +/- to look up or down

32.49 fps (600x600), X8R8G8B8 (D16)
HAL (pure hw vp): RADEON 9800 PRO
Use arrow keys or joystick to move around
Use numpad +/- to look up or down

Figure 4. *Our framework seamlessly integrates standard triangle rendering with ray-casted volume.*

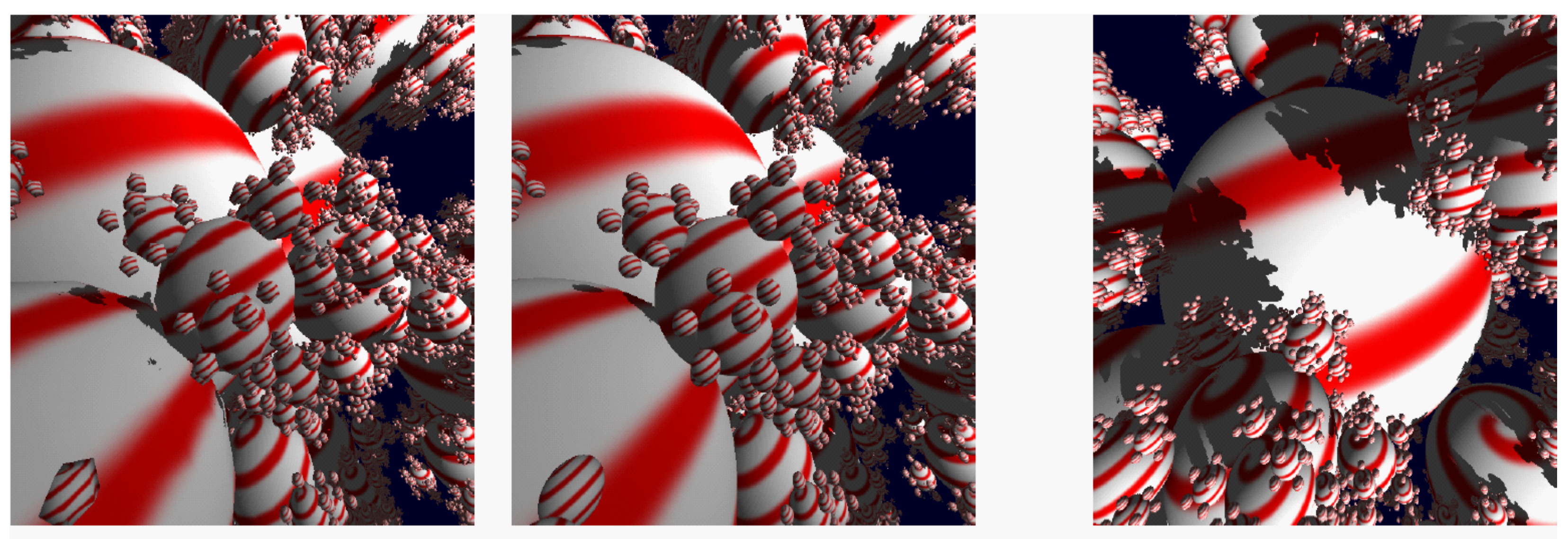


Figure 5. *Sphereflake rendering: The left image rendered using standard rasterization. The center and right images rendered with our hardware ray-caster. The right image uses shadow mapping technique.*

# Spheres	# Poly	Rasterizer (FPS)	RayTracer (FPS)	SpeedUp
1	1680	94	90	-4%
10	8880	69	64	-7%
91	38174	53	48	-9%
829	161646	36	37	3%
7381	581550	20	28	40%
66430	1762530	9	18	100%

Table 1: *Performance of our renderer vs. standarad hardware renderer in rendering the snowflake.*