GPU Objects

Sunil Mohan Ranta, Jag Mohan Singh, and P.J. Narayanan

Center for Visual Information Technology
International Institute of Information Technology
Hyderabad, India
{smr@research., jagmohan@research., pjn@}iiit.ac.in

Abstract. Points, lines, and polygons have been the fundamental primitives in graphics. Graphics hardware is optimized to handle them in a pipeline. Other objects are converted to these primitives before rendering. Programmable GPUs have made it possible to introduce a wide class of computations on each vertex and on each fragment. In this paper, we outline a procedure to accurately draw high-level procedural elements efficiently using the GPU. The CPU and the vertex shader setup the drawing area on screen and pass the required parameters. The pixel shader uses ray-casting to compute the 3D point that projects to it and shades it using a general shading model. We demonstrate the fast rendering of 2D and 3D primitives like circle, conic, triangle, sphere, quadric, box, etc., with a combination of specularity, refraction, and environment mapping. We also show combination of objects, like Constructive Solid Geometry (CSG) objects, can be rendered fast on the GPU. We believe customized GPU programs for a new set of high-level primitives – which we call *GPU Objects* – is a way to exploit the power of GPUs and to provide interactive rendering of scenes otherwise considered too complex.

1 Introduction

Points, lines, and polygons are the basic primitives in conventional graphics. Acceleration hardware is optimized to process them quickly in a pipeline. Complex shapes are converted to these primitives before rendering. Procedural geometry, on the other hand, involves on-the-fly creation of arbitrarily accurate shape from compact descriptions, usually in the form of implicit equations. The graphics display pipeline cannot render procedural geometry directly. Procedural objects are converted to piecewise linear models using polygons and lines before rendering. This results in a loss in resolution and incurs computational overhead. Ray-tracing methods can handle procedural geometry to produce high-quality renderings. These methods have very high computational complexity and are not suitable for interactive applications.

The Graphics Processor Units (GPUs) have seen very steep growth in processing capabilities. They deliver highest computation power per unit cost today and have been improving at a quick pace. Introduction of programmability in GPUs at the vertex and the fragment levels has brought novel uses of the graphics hardware. We present several examples of fast and accurate rendering of procedural objects on the GPUs in this paper. The equations of the objects are evaluated exactly at each pixel it projects to in a way similar to the ray-tracing techniques. This results in high quality rendering at all resolution levels and exact, per pixel lighting. We also apply the technique to objects that are traditionally not considered to be procedural. All of this is performed at interactive frame rates.

Procedural geometry has many benefits over polygonal geometry. In the latter, the surface is approximated by a triangle mesh. Triangulation in itself is an overhead, which requires time consuming preprocessing of the geometry. Triangulated mesh requires high memory bandwidth from the CPU to the GPU and huge video memory for storage. Procedural geometry can save both bandwidth and memory requirements drastically. Resolution independent rendering of curved surfaces [1] was achieved using procedural geometry on GPU. Resolution independence results in the curved surfaces appearing exactly curved at all magnification levels. Procedural geometry finds its application in Constructive Solid Geometry (CSG), which is used in solid modeling to create complex shapes by combining simple shapes primitives with boolean operators on sets [2]. The primitives used in CSG are ground set of shapes such as box, sphere, cylinder, cone, torus, prism, etc.

In this paper, we outline a general procedure for rendering a wide class of objects using ray-casting from a GPU. We also show how high-quality lighting options can be computed exactly for these objects. We demonstrate the procedure to interactively render several generic objects very fast on the GPU. These include triangle, quadrilateral, circle, conic, sphere, box, tetrahedron, quadric, etc. We also show how different lighting models can be incorporated into the rendering. We then extend the basic procedure to render a combination of objects together and demonstrate it on various CSG objects. We believe this is the first time high-quality ray-casting of CSG objects has been performed at interactive rates. Our work lays the foundation for a class of *GPU Objects* that can be rendered interactively in high quality. We show overview of our results in

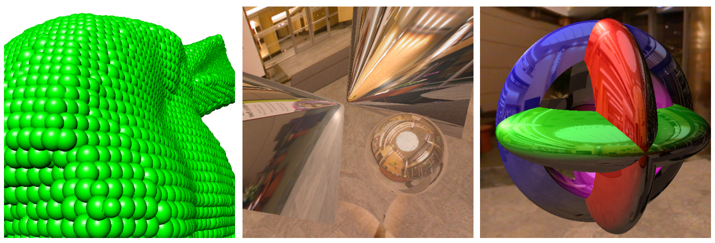


Fig. 1. GPU Objects: (a) Bunny with 36K spheres at 57 fps. (b) Hyperboloid with reflection and sphere with reflection and refraction at 300 fps. (c) Four-primitive CSG at 22 fps.

Figure 1. A GPU Object consists of program fragments for the CPU, the vertex shader and the pixel shader and can be called like a function by an application program with the parameters for the objects. As the limitations of today's GPUs go away and the architecture evolves to include more memory, longer programs, and more flexible shaders, customized GPU Objects will be a way to exploit their compute power to provide interactive rates to rich scenes.

2 Prior Work

Graphics hardware is getting faster which helps rasterization to produce good rendering effects. The effects generated by ray tracing are the most realistic. Programmable graphics hardware is able to deliver the promise of realistic rendering effects using ray tracing. Ray tracing of procedural objects [3] transforms the three dimensional ray-surface intersection into a two dimensional ray-curve intersection problem, which is solved by using strip trees. It was demonstrated on procedural objects such as fractal surface, prisms and surfaces of revolutions. Ray tracing of complex scenes [4] incorporated several new schemes such as bounding box being a good convex hull of the object, division of the object into hierarchies and efficient data structure for traversing this hierarchy. These techniques were used for speeding up ray tracing. Ray-tracing on GPU [5] with different methods such as Whitted ray tracing, path tracing, and hybrid rendering algorithms showed that it runs faster on GPU than on CPU. Ray Engine [6] does a ray-triangle intersection on GPU and achieves effects such as recursive ray tracing, path tracing, form factor computation, photon mapping, subsurface scattering, and visibility processing. Ray tracing of ellipses with EWA filtering which results in anti-aliased splats was done by Botsch et. al [7]. Ray tracing of perspectively correct ellipsoids on GPU [8] render ellipsoid by transformation of a unit sphere. GPU accelerated primitives [9] presents a framework for rendering of quadric surfaces on GPU. They use a different Ray Tracing Area for each type of quadric to minimize the load on pixel shader. Ray tracing of quadrics on GPU [10] which uses efficient bounding box computation has been done recently. Fully Procedural Graphics [11] proposes the extension of graphics hardware so that it may be able to support procedural rendering of objects.

CSG has already been a well explored area using CPU based algorithms. Initial methods included the generalization of scanline algorithm using ray tracing for rendering of intersecting objects [12]. CSG graph representation is optimized into Convex Difference Aggregates for efficient CSG of convex objects [13]. Normalization and bounding box pruning for CSG [14] demonstrated on the pixel planes architecture and surface intersection using bounding box optimization [15] achieve faster CSG. The use of stencil buffer and depth peeling techniques was done for CSG by Guha et. al [16]. Blister [17] evaluates Blist representation of CSG expression directly on GPU and is able to render large number of primitives in real-time.

3 Rendering Geometric Objects

The fundamental operation used by our rendering method is the intersection of a ray with an object. The intersection is computed in the pixel shader for each pixel, given the parameters of the object being rendered. This step is essentially the conventional ray casting implemented on the pixel shader. The points on the ray in parametric form can be represented as $P = O + tD$, where t is the parameter along the ray, O the camera center and D the direction of the ray. The intersection of the ray with an object given by $f(P) = 0$ can be calculated by substituting the parametric form for P and solving the resulting equations for the smallest value of t. Polynomial forms of $f()$ span a range of useful objects and are easy to solve. Many non-linear forms of $f()$ can also be solved for analytically. Representations such as triangle, quadrilateral, box, tetrahedron, etc., are not procedural, but can be intersected with a ray efficiently on the pixel shader.

We now give a generic procedure to draw a general object using an appropriate shader. The object is given by the implicit form $f(P) = 0$.

Algorithm 1. renderGeomObject(f)

```
    CPU: An OpenGL program performs the following.
 1: Pass the parameters of $f()$ to the graphics pipeline as graphics bindings such as
    texture coordinates, color and position. A texture can be used if more data needs
    to be sent.
 2: Draw an OpenGL primitive such that the screen-space area of the object is covered
    by it. This ensures that all pixels will be drawn and the corresponding shaders will
    be executed. The primitive used could be a dummy one with the right number of
    vertices.
    Vertex Shader: A vertex shader performs the following.
 3: Pass the parameters from the CPU to the pixel shader.
 4: Transform the OpenGL primitive drawn by the CPU to cover the screen-space area
    of the object using the object parameters.
 5: Perform other pixel independent calculations required for the pixel shader and
    passes on the results.
    Pixel Shader: A pixel shader performs the following.
 6: Receive the parameters of the object and own pixel coordinates $(i, j)$ from the
    pipeline.
 7: Perform an acceptance test for $(i, j)$ based on the parameters of $f()$. This involves
    computing exactly if the pixel will be on the projected region of the object. This
    may require the parameters of $f()$, the Modelview, Projection, Viewport matrices,
    etc. The acceptability can be computed in a 2D texture space in some cases.
 8: Compute the ray-object intersection for accepted pixels. This involves solving an
    equation in $t$ that is based on $f()$.
 9: Compute the 3D point corresponding to the smallest $t$ among the intersecting
    points. Also compute the depth and normal at that point using $f()$.
10: Shade the pixel using the lighting, material, normal, and viewing information that
    is available to the shader. The reflected ray at the intersection point can be pursued
    to apply environment mapping, refraction etc.
```

It is important to setup the screen-space bounding area as compactly as possible as it affects the computation time. A compact bounding polygon is a good option. The CPU and the vertex shader set this up in combination. The interpolation of texture coordinates performed by the primitive assembly unit can be exploited to send values to all pixels, if suitable. This would be useful for data like the 3D position, depth, etc., that may be needed at the pixel shader. It is also possible to draw a single point-primitive with appropriate point-size [10]. This can involve extra calculations performed at the pixel shader. The pixel shaders code memory and computation time could be stretched by this, while the task of the CPU and the vertex shader are simplified. Every pixel in the bounding area need to check if it is part of the actual object. The ray-object intersection will give imaginary results for pixels that are outside the object. Easier acceptance tests may be available for some shapes. The intersection point for the accepted rays has 3D position (from the ray equation), a normal vector (from derivatives of $f()$), and a view direction (from the camera position). Every pixel can be lit accurately using these. The reflected and refracted rays can be computed and used for effects like environment mapping and refraction. Recursive ray tracing is, however, not possible as the pixel shaders don't support recursion or deep iteration due to the SIMD programming model available at the fragment units.

We now explain how the above generic procedure can be used to render several different 2D and 3D objects.

3.1 Planar Shapes

We consider the shapes circle, conic, triangle, and parallelogram. For planar shapes, the pixel acceptance can be performed in two ways: in the 3D space and in the texture space. In the former, the ray-plane intersection and the acceptance tests are performed in 3D space. In the latter, the vertex shader converts the coordinates to in-plane coordinates and passes them as texture coordinates. These values are interpolated by the rasterizer. The pixel shader performs the acceptance test using the 2D equations using the interpolated texture coordinates. Texture space acceptance test is more efficient but requires a dedicated bounding area. This means only one primitive can be rendered at a time.

Circle: A square is used as the bounding area for the circle. A more close fitting regular polygon can also be used, but at the cost of increasing the vertex shader time. The parameters for the circle are its center, radius and the plane normal. These values are passed using texture coordinates to the shaders. A dummy square with coordinates $(\pm 1, \pm 1)$ is passed by the CPU and are transformed by the vertex shader to a square with length twice the radius. The implicit equation $|P - C| - r \leq 0$ is evaluated in world coordinates to check validity in 3D space. For texture space calculations, the in-plane coordinates of the square corners are sent by the vertex shader as a texture coordinate. This is interpolated by the rasterizer and the interpolated value is available to the pixel shader. The circle equation can be evaluated in 2D using the texture coordinates. We illustrate the algorithm for rendering of a circle with environment mapping below :

Algorithm 2. CircleRender(Center, Radius and Normal)

```
1: CPU  Send a dummy quad with coordinates $(\pm 1, \pm 1)$.
2: Vertex Shader  Convert the corner coordinates of the quad to in-plane coordinates and set them as texcoord.
3: Pixel Shader  receive center and in-plane coordinates
4: if distance of current pixel from center > radius then
5:     discard
6: else
7:     use normal for lighting.
8:     use reflected ray for environment mapping. reflected ray is obtained by reflecting the ray from camera center to current pixel about the normal
9:     return color and depth of accepted pixel
10: end if
```

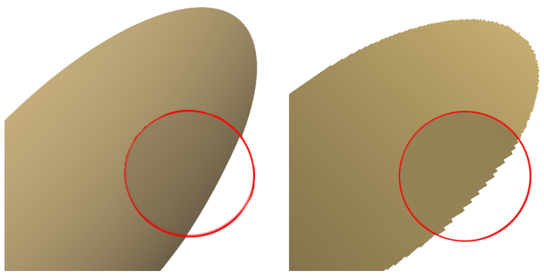


Fig. 2. (a)Ellipse rendered procedurally on GPU (b) Ellipse rendered using a texture of 512x512 resolution. GPU based rendering is resolution independent and has no aliasing artifacts where circle shows the zoomed view.

Conic: Conic is a curve formed by intersecting a cone with a plane. Shapes such as hyperbola, ellipse, circle, and parabola can be represented using conics. Conics are represented in matrix notation as $PCP^T = 0$ where C is a symmetric matrix. A conic is described using 6 parameters for C, and the base plane normal. The bounding area for a conic is computed by finding orthogonal lines tangent to the conic. The bounding area thus formed is a rectangle. The dummy square from the CPU is aligned to this rectangle by the pixel shader. The shader also computes the in-plane coordinates of the rectangle vertices as texture coordinates, which are interpolated before reaching the pixel shader. Pixel shader evaluates PCP^T for the in-plane coordinates and its sign is used for acceptance.

In fragment shader we first compute intersection of the ray with plane. Equation of plane with normal n and passing through point p is given by $n \cdot (p - x) = 0$ and its intersection with ray is given by $t = n \cdot (p - O)/n \cdot D$. The traced point is converted to 2D point and then checked with equation of the conic. For texture space test, the ray-plane intersection and conversion to 2D point is computed in vertex shader and interpolated values are used in pixel shader for acceptance test using sign of PCP^T.

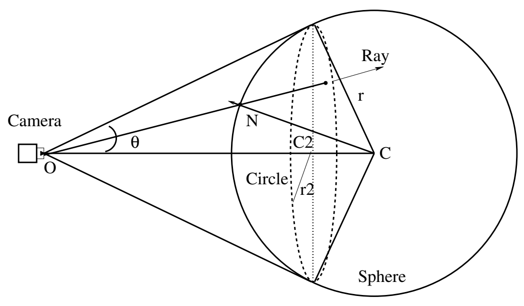


Fig. 3. The sphere is converted into a circle normal to camera view vector

Triangle and Parallelogram: Triangle is strictly not a procedural object. However, the interior of the triangle is given by $0 \leq u, v \leq 1$, where u, v are the barycentric coordinates calculated using the vertices. The bounding area for triangle is a one pixel wider triangle in screen space. Three points are sent from the CPU which are converted to the barycentric coordinates by the vertex shader using Möller et al. algorithm [18]. Triangle bound checking on barycentric coordinates is used for acceptance test. For 3D space test the coordinates are evaluated in pixel shader and then used for bound checking. The same can be achieved in texture space, by evaluating the barycentric coordinates on the vertex shader and interpolating it to the pixel shader.

A parallelogram can be handled in a similar way. The condition for acceptance is $0 \leq u, v \leq 1$. As for triangle, this acceptance test can be done either in 3D space or in texture space.

3.2 3D Shapes

We consider the following 3D shapes: sphere, quadric, cylinder, cone, parallelepiped, and tetrahedron. For 3D objects, the bounding area is either a bounding box for the object in 3D space or bounding rectangle in projected space. 3D shapes can have one or more intersections with ray. Nearest intersection is used for calculating depth and shading. For use as CSG primitive all intersections are important.

Sphere: A sphere can be represented using a quadric. It is handled more efficiently than a general quadric by Toledo et al. [9]. We use a different approach in order to render it even more efficiently by reducing the problem to rendering of a circle of appropriate radius and orientation. Figure 3 shows a sphere with center C and radius r and its projected circle with center $C2$, radius $r2$, and oriented along the ray from sphere center to camera center. The procedure for rendering a sphere, SphereRender(C, r) is described in Algorithm 3.

Thus, we reduce the bounding area for sphere to bounding area of the circle, which in general is a camera facing regular polygon. It can be drawn using a polygon with optimal edges or even as a single point with proper size. Figure

Algorithm 3. SphereRender(C, r)

```
1: CPU  Send a dummy quad
2: Vertex Shader  $r2 = r\cos(\sin^{-1} r/d)$; $C2 = (1 - \frac{r^2}{d^2})C + \frac{r^2}{d^2}O$.
   Convert to in-plane coordinates using [18] and send corner points as texcoord.
3: Pixel Shader  receive $C, C2, r$ and $r2$ and in-plane coordinates $P : (u, v)$
4: if $|P - C2| > r2$ then
5:   discard
6: else
7:   solve quadratic equation for ray-sphere intersection and use smaller $t$.
8:   light using 3D point, normal, and view vector.
9:   use reflected ray for environment mapping if enabled.
10:  if refraction then
11:    intersect refracted ray again with sphere,
12:    refract it once more and use it for environment mapping.
13:  end if
14:  set color of pixel as linear combination of above colors.
15:  return color and depth of accepted pixel
16: end if
```

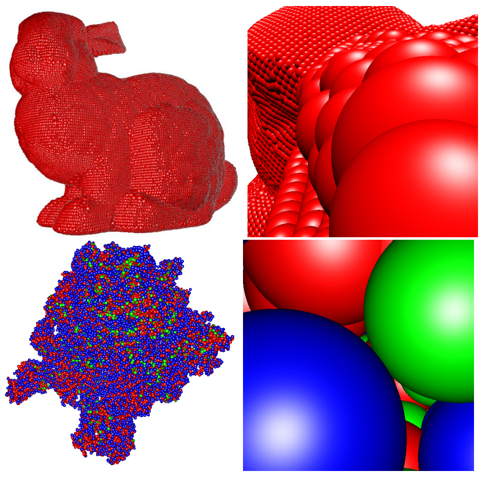


Fig. 4. Top: Bunny Model with 35,947 spheres is rendered at 57 fps at 512x512 viewport and its zoomed view. Bottom: Ribosome molecule with 99,130 spheres is rendered at 30 fps at 512x512 viewport and its zoomed view.

4 shows rendering of large datasets represented as collection of spheres at interactive frame rates. We used NVIDIA GeForce 6600 GT in our experiments.

Cylinder and Cone: The bounding area for cylinder is billboard rectangle along cylinder axis and a square at end of the cylinder facing the camera. The bounding area for cone is billboard triangle along cone axis and a square at the base of the cone. Ray intersection involves solving quadratic equation, and real roots producing pixels are accepted.

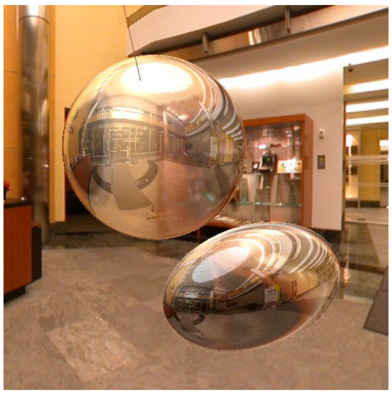


Fig. 5. Sphere and Ellipsoid with environment mapping and refraction at 300 fps

General Quadric: Quadric surfaces are second order algebraic surfaces represented in matrix notation as $PQP^T = 0$ where Q is a symmetric matrix and P is a point. Bounding area for the quadric is computed from its conic projection defined as $C = PQP^T$ where P is projection matrix. The base plane of the conic is QC. Ray intersection of quadric is given by roots of quadratic equation in t and pixels resulting in complex values of t are discarded. For texture space acceptance test, ray is intersected with base plane of conic in vertex shader and texture space values of intersections are used in pixel shader for inside-conic test. Quadrics with reflection, refraction and environment mapping are shown in Figure 5.

Parallelepiped and Tetrahedron: Parallelepiped is formed by three pairs of parallel parallelograms. A parallelepiped can be represented using four vertices. The six parallelograms can be described using these vertices and the intersection with each is computed. Bounding area for parallelepiped is given by three parallelogram faces. 3D space ray-parallelogram intersection is computed for every face, and the nearest intersection point is considered for lighting. For CSG both intersection are of importance.

Tetrahedron is formed by four triangles and can be represented using its four vertices. The four triangles can be described using these vertices. A regular tetrahedron is represented using apex position, direction vector and side length. Bounding box for tetrahedron is formed by four triangles of it and the intersection with each triangle is computed using ray-triangle intersection in 3D space.

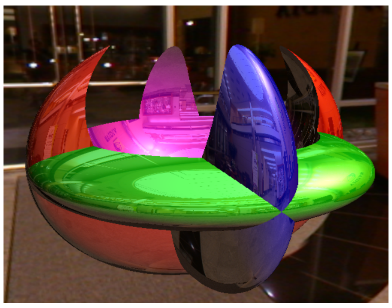
4 Rendering CSG Objects

We showed how different objects can be rendered fast using special shaders on the GPU. The object is rendered with correct depth and color values. Thus, the GPU rendering can be mixed with normal polygonal rendering and the picture will have correct occlusions and visibility. We now see how a combination of objects can be drawn together by the GPU. The motivation is to draw CSG objects, which are formed using union, intersection, and subtraction of other objects. CSG objects are represented using CSG trees of primitives and are popular in CAD to describe objects exactly. Procedural objects are commonly used in CSG.

We show the rendering of CSG objects that are boolean combinations of the objects we have seen earlier. Ray casting at the pixel shader is used for this.

Algorithm 4. renderCSGObject()

```
CPU:
1: Write the primitives of the CSG tree into the texture memory with appropriate
   descriptions.
2: Calculate the screen-space bounding area for the positive primitives in the scene
   and draw it using OpenGL.
Pixel Shader:
1: Read the CSG tree and information about the primitives from the texture.
2: Calculate all ray intersection for every primitive.
3: Sort the intersections by $t$. Preserve primitive id for each intersection point. Set $toi$
   as +1 for entry intersections (smaller $t$) and -1 for exit intersections (larger $t$).
4: Create two counters: $plus$ for positive and $minus$ for negative primitives and ini-
   tialize both to zero.
5: Examine each intersection point. Add its $toi$ to the $plus$ counter if the corresponding
   primitive is a positive one. Add $toi$ to the $minus$ counter otherwise. The counters
   contains the number of positive and negative objects encountered by the ray from
   beginning.
6: Stop when the $minus$ counter is zero and $plus$ counter is positive. This is the
   first visible point along the ray. The primitive for this intersection is the visible
   primitive.
7: Compute depth and normal using the visible primitive. Reverse normal direction
   if the visible primitive is negative.
8: Light the point using the normal material properties etc.
```


(a)

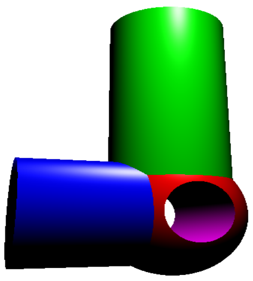

(b)

Fig. 6. (a) CSG of four quadrics with reflection and environment mapping at 20 fps. (b) CSG of cylinders and spheres with phong shading at 20 fps.

However, the ray could intersect multiple objects and the boolean combination between them decides what actually gets drawn. Thus, all objects in the CSG tree need to be rendered *together* to generate the correct image at each pixel. Points are evaluated for being on the boundary of composite object for drawing [17].

Our scheme stores the CSG tree in texture memory and its reference is made available to the shaders. Each primitive is represented using an id for its type, the parameters for that type of primitive, and a flag to indicate if the primitive is used in an additive or subtractive sense. We outline a procedure renderCSGObject() (Algorithm 4), to draw a simple CSG object, consisting of a set of positive primitives and a set of negative ones.

The above procedure can render complex CSG objects. Rendering many primitives together is a challenge on today's GPUs with its limitations on the shader length. We are able to render CSG objects shown in Figure 6 containing upto 5 quadric primitives on the NVidia 6600GT system. This will improve with newer generation cards and very complex CSG objects will be possible in the future. We show some of our results in the accompanying video.

5 Conclusions and Future Work

In this paper, we presented a scheme for rendering several high-level objects using appropriate shaders on programmable GPUs. We showed interactive rendering of several geometric and CSG objects with sophisticated, per pixel, lighting. The figures and accompanying video demonstrate the effectiveness and speed of our method in rendering many high level objects.

The GPUs are getting more powerful and more programmable with every generation. While they speed up the rendering of conventional geometry, their impact can be felt more in rendering higher level primitives that are slow to render today. This can be made possible using specialized shader packages that can draw certain types of objects quickly. These packages – which we call *GPU Objects* – could be parametrized to generate a class of objects and a class of rendering effects. These GPU Objects can be invoked by a rendering program as they do with OpenGL primitives. They can be mixed freely with one another and with conventional geometry rendering and will produce the correct visibility and lighting effects. We are currently devising generic GPU Objects that can be parametrized to get a variety of objects. Such objects will be possible to render at high speeds on the future GPUs as they get more flexible.

References

1. Loop, C.T., Blinn, J.F.: Resolution independent curve rendering using programmable graphics hardware. ACM Trans. Graph. **24** (2005) 1000–1009
2. Requicha, A.A.G.: Representations for rigid solids: Theory, methods, and systems. ACM Comput. Surv. **12** (1980) 437–464
3. Kajiya, J.T.: New techniques for ray tracing procedurally defined objects. ACM Trans. Graph. **2** (1983) 161–181

4. Kay, T.L., Kajiya, J.T.: Ray tracing complex scenes. In: SIGGRAPH '86. (1986) 269–278
5. Purcell, T.J., Buck, I., Mark, W.R., Hanrahan, P.: Ray tracing on programmable graphics hardware. In: SIGGRAPH '02. (2002) 703–712
6. Carr, N.A., Hall, J.D., Hart, J.C.: The ray engine. In: HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Eurographics Association (2002) 37–46
7. Botsch, M., Hornung, A., Zwicker, M., Kobbelt, L.: High quality surface splatting on today's gpus. In: Proc. of symposium on Point-Based Graphics '05. (2005) 17–24
8. Gumhold, S.: Splatting illuminated ellipsoids with depth correction. In: VMV. (2003) 245–252
9. Toledo, R., Levy, B.: Extending the graphic pipeline with new gpu-accelerated primitives. Tech report, INRIA (2004)
10. Christian Sigg, Tim Weyrich, M.B., Gross, M.: Gpu-based ray-casting of quadratic surfaces. In: Proceedings of Eurographics Symposium on Point-Based Graphics 2006 (to appear). (2006)
11. Whitted, T., Kajiya, J.: Fully procedural graphics. In: HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. (2005) 81–90
12. Atherton, P.R.: A scan-line hidden surface removal procedure for constructive solid geometry. In: SIGGRAPH '83. (1983) 73–82
13. Rappoport, A., Spitz, S.: Interactive boolean operations for conceptual design of 3-d solids. In: SIGGRAPH '97. (1997) 269–278
14. Goldfeather, J., Monar, S., Turk, G., Fuchs, H.: Near real-time csg rendering using tree normalization and geometric pruning. IEEE Comput. Graph. Appl. **9** (1989) 20–28
15. Mazzetti, M., Ciminiera, L.: Computing csg tree boundaries as algebraic expressions. In: SMA '93: Proceedings on the ACM symposium on Solid modeling and applications. (1993) 155–162
16. Guha, S., Mungala, K., Shankar, K., Venkatasubramanian, S.: Application of the two-sided depth test to csg rendering. In: I3D, ACM Interactive 3D graphics. (2003)
17. Hable, J., Rossignac, J.: Blister: Gpu-based rendering of boolean combinations of free-form triangulated shapes. ACM Trans. Graph. **24** (2005) 1024–1031
18. Moller, T., Trumbore, B.: Fast, minimum storage ray-triangle intersection. J. Graph. Tools **2** (1997) 21–28