Implementing a Rasterization Framework for a Black Hole Spacetime

YOSHIYUKI YAMASHITA1,a)

Received: October 26, 2015, Accepted: April 5, 2016

Abstract: The theory of general relativity predicts that the strong gravity of a black hole bends the trajectories of light rays. Calculating their bendings numerically, we can obtain a 3D CG image when the view point is set in the black hole spacetime. The existing researches adopt the ray-tracing method for rendering while we adopt the rasterization method in this paper. In order to achieve fast perspective projection in the curved spacetime, we calculate more than thirty million light trajectories on an optimally constructed computational mesh in advance and let a GPU interpolate them when rendering. Furthermore, in order to render the lines and triangular polygons of CG objects accurately, we apply the dynamic subdividing technique (tessellation). Various types of CG programs can be easily written in the same way as in the conventional 3D CG programming with a common graphics API. Utilizing the recent computing power of the GPU, the rendering performance of nearly one million polygons per second is achieved even on a notebook PC.

Keywords: rasterization, black hole, graphics processing unit

1. Introduction

The theory of general relativity that Einstein proposed predicts that the strong gravity of a black hole bends the trajectories of light rays[1], [2]. Calculating their bendings numerically, we can naturally extend the concept of the usual three dimensional computer graphics (3D CG in below) into the black hole’s spacetime. Let us call such an extended CG as the general-relativistic 3D CG. One of its visual effects is known as a gravitational lens effect[3], which has been studied by a number of research papers (e.g., Refs. [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]).

The general-relativistic 3D CG has two kinds of purposes.

One is to contribute to the study of astrophysics, in particular, the study of high energy radiation near around a black hole[4], [6], [7], [13], [14]. In this case the numerical simulation is the main theme to verify with the actually observed data obtained from radio telescopes. Therefore it is almost sufficient to render a simple geometrical shape of a CG object just like a plane, disk, torus, or sphere which represents a model of an accretion disk[3] around the black hole. Due to its simplicity, some of the researches[7], [13] have achieved on-the-fly realtime CG image generation of the gravitational lens effect by utilizing the massive computing power of a recent graphics processing unit (GPU in below).

The other purpose of the general-relativistic 3D CG is to contribute to the field of physics education and/or entertainment that gives an intuitive understanding to high school/junior high school students and amateur scientists who have a particular interest in theory of general relativity[5], [8], [9], [10], [11], [12]. In this case, in contrast to the former case, complex CG objects like space ships, rockets, and so on are preferable for the students and amateurs to feel familiar and intuitive. In the case of conventional 3D CG it is quite common to define the shapes of such space ships and rockets with more than a thousand triangular polygons and render them on GPU by the rasterization framework with common graphics APIs like OpenGL[15] and DirectX[16]. In the general-relativistic 3D CG, in contrast, no research has ever achieved such rasterization based rendering. We challenge it in this paper and fully exploit the graphics processing power of the GPU.

To the author’s best knowledge, all the existing research adopt the ray-tracing technique/method but not the rasterization technique/method due to the following two reasons. (1) The calculation cost of on-the-fly perspective projection in the black hole spacetime is extremely high. (2) A straight line in the black hole spacetime is not perspective projected to a straight line but a curved line in general. In the same sense, the three straight lines of a triangular polygon are not projected to straight lines in general. We solve the problem (1) by constructing a well-formed computational mesh and calculating the light trajectories at all the crossover points on the mesh in advance. The calculation results are loaded to GPU memory and the vertices of CG objects are linearly interpolated by the GPU hardware. Furthermore we solve the problem (2) by applying the dynamic, recursive subdividing technique, often called tessellation, to lines and polygons in the same way as in Refs. [17], [18], [19].

Our system in this paper consists of two stages (see Fig. 1). The first half is the preprocessing stage to prepare the mesh data. The last half is the actual rendering stage, in which the host program is implemented with OpenGL API and the shader programs.
on a GPU are written in OpenGL Shading Language [20] (GLSL in below). Because of adopting OpenGLE/GLSL framework, we can manage the CG primitives in the same way as in the conventional 3D CG programming. In fact we will show various kinds of CG programs which render point sprites, wireframes, and the 3D modeling data defined by more than a thousand of polygons in the black hole spacetime. The programs can achieve the rendering performances of more than one million polygons per second on the desktop PC, Mac Pro, and nearly one million polygons per second on the notebook PC, MacBook Pro.

The remainder of this paper is organized as follows. In Section 2 we discuss our framework (geometry model, rendering method, and color model) that we assume when rendering a CG scene in the black hole spacetime. In Section 3 we explain how to perform the perspective projection by utilizing the trilinear interpolation hardware of a GPU. In Section 4 we actually render the OpenGL primitives: point sprites, lines, and triangular polygons. In particular, lines and polygons are dynamically subdivided to render their distorted appearances accurately. We also discuss the rendering speeds.

2. Framework

In this section we discuss our framework of the general-relativistic 3D CG and the two key problems of the rasterization which we must solve.

2.1 Geometry Model

Figure 2 illustrates the geometrical model in this paper, where the gray disk with white letters “BH” in the figure represents the black hole. A light ray (each of the solid and dashed arrows in the figure) is assumed to be emitted from a point $P$ on a polygon surface of a self illuminating CG object, or emitted from a certain light source and then reflected at the point $P$. Such a ray passes near a black hole, then proceeds through a pixel point $S$ on a perspective screen, and then finally reaches a view point $V$. Because the ray bends by the strong gravity of the black hole, there can be more than one such ray that goes through both $P$ and $V$. In Fig. 2 there are the two rays of solid and dashed arrows that go through the points $S$ and $S'$, respectively. Note that, if the light source exists, the ray trajectory from the light source to $P$ (the gray bold arrow in Fig. 2) also bends near the black hole. However, in order to reduce the calculation cost, we simply assume that the line segment of such a ray from the light source to $P$ does not bend and that the polygon is rendered according to Phong’s shading model [21] with an ideal parallel light source at infinity.

We fix the black hole at the origin of the world coordinate system (or global coordinate system). In this sense, we implicitly identify the world coordinate system with the black hole space-time coordinate. The type of the black hole we discuss here is the simplest, spherically symmetric one, called Schwarzschild black hole, which is defined by the following Riemann metric [1], [2]

$$ds^2 = \left(1 - \frac{a}{r}\right)dt^2 - \left(1 - \frac{a}{r}\right)^{-1}dr^2 - r^2(d\theta^2 + \sin^2 \theta d\phi^2) \tag{1}$$

where the triplet $(r, \phi, \theta)$ expresses the polar space-like coordinates and $t$ the time-like coordinate. The constant $a$, called a gravitational radius, defines the radial boundary between the inside $(r < a)$ and outside $(r > a)$ of the black hole, where we consider only the outside for the view point and polygon points. The trajectory of every light ray is defined by a set of second-order, nonlinear, ordinary differential equations, called a geodesic equations, derived from the Riemann metric according to the theory of general relativity (see the details in Ref. [1]). In our case of Eq. (1), because of the symmetricity of the black hole, we can omit the coordinate $\theta$ by fixing $\theta = \pi/2$ so that, without the loss of generality, the ray trajectory can be defined by the following set of three equations,

$$i = -\frac{a}{r^2} \left(1 - \frac{a}{r}\right)^{-1} \dot{r}$$

$$\ddot{r} = -\frac{1}{2} \left(1 - \frac{a}{r}\right) \left(\frac{a^2}{r^2} - \frac{a^2}{(r-a)^2} - 2r \dot{\phi}\right) \tag{2}$$

where the dot ‘$\dot{}$’ and the double dot ‘$\ddot{}$’ mean the first and second order derivation operators $d/dt$ and $d^2/dt^2$, respectively, with an arbitrary parameter $\lambda$. There are two kinds of methods available to solve Eq. (2). One method is to use the fourth-order Runge-Kutta integration as many existing research do [4], [7], [8], [9], [10]. The other method is to use the analytic solution [13].

2.2 Problems of Rasterization

To the author’s best knowledge, all research up to now adopt
the ray tracing method \(^*1\), which traces the light ray trajectory in Fig. 2 back in time. Mathematically speaking, the ray tracing method must solve the \textit{initial value problem} (IVP in below) of Eq. (2), that is, given an arbitrary view point \(V\) and an arbitrary pixel point \(S\) (or \(S'\)) on the perspective screen, we first calculate the initial point and initial direction of the light ray, then trace the ray trajectory according to Eq. (2), and obtain the point \(P\) on a certain polygon surface if there exists such a surface that the ray hits on. This calculation cost is expensive but, with the computing power of a recent GPU, several research effort/paper\[^7\], \[^13\] have accomplished solving the IVP on the fly and generating CG images in realtime. However they treat a considerably simple geometrical shape of a CG object like a plane or a disk. The rendering programs are specialized to their specific CG objects. When challenging to render more complicated objects that may be defined by more than a thousand of polygons, it seems still hard to render them in realtime by the ray tracing method.

In contrast to the existing researches, we adopt the rasterization method which includes no multiple trace of reflective and refractive rays. For these reasons, we propose another method of utilizing the hardware linear interpolation unit of a GPU, though the idea is not new at all. Our system consists of the preprocessing stage and the rendering stage, as illustrated in Fig. 1. In the preprocessing stage we first construct a computational mesh, then solve the BVP for every mesh point, and store the calculation results (a set of BVP solutions) into a hard disk. Next, in the rendering stage, for given arbitrary \(V\) and \(P\), the solution is linearly interpolated from the pre-calculated solutions loaded from the hard disk. Now arises the question whether the interpolation error can be sufficiently small or not. The answer is yes when we prepare enough number of solutions on a deliberately constructed mesh. The rendering cost is expected to be particularly smaller than those of the above two on-the-fly methods because we do not solve the BVP on the fly and use the GPU hardware effectively. The key point is how to construct an optimal structure of computational mesh. We will discuss the details of mesh construction in Section 3.

The other technical problem of the rasterization is that a straight line in the curved spacetime is perspectively projected to a curved line but not a straight line in general due to light ray bending. Thus we cannot use the common rasterization method. To solve this problem, we apply the subdividing technique that divides a given line into two or more shorter lines recursively and then rasterizes the lines. If the divided lines are shorter enough such as the approximation errors are on the sub-pixel scale, the human eye cannot recognize the errors. The same technique is also applicable to a triangular polygon. In this paper we implement this technique on the geometry shader of a GPU (see Fig. 1). The details are explained in Section 4.

Here we put one more assumption that any polygon point \(P\) stands at rest in the black hole spacetime or moves slowly enough, because it is too complex to trace \(P\) when \(P\) moves with semi-light speed. In contrast, the view point can move at any speed if the speed is not beyond the light speed. In this case it is enough to insert the Lorentz transformation (see Ref. [14] from the view point of the 3D CG on special relativity) after the usual viewing transformation and before perspective projection over the whole sequence of 3D CG coordinate transformations.

\[\text{2.3 Color Model}\
\]

One of the visual effects necessary for the relativistic 3D CG is the red/blue-shift of light frequency due to gravitational potential and/or the Doppler shift of light. In order to implement this effect simply in a CG program, we assume that the rendering color of every CG object is approximately represented with the color temperature of an ideal black body radiation. Because it is known that the ratio of the light frequency change is equivalent to that of the temperature change in the case of black body radiation\[^{[22]}\], given the original temperature \(T\) of the CG object and the ratio \(f\) of the light frequency change (see Section 3.4), the temperature \(T'\) that the view point observes is simply calculated as \(T' = fT\), and further translated into the RGB color \((r', g', b')\) on the perspective screen.

3. Fast Perspective Projection by Trilinear Interpolation

In this section we first clarify our requirements for the perspective projection in the black hole spacetime. Next, we optimize the structure of the computational mesh for interpolating the perspective projection.

\[\text{3.1 Requirements for Perspective Projection}\
\]

Due to the spherical symmetry of the black hole, every light ray we discuss here travels along the plane that contains the view point \(V\), the center \(O\) of the black hole, and the polygon point \(P\). Thus it is enough to treat the perspective projection only on this plane.
Now let us consider a light ray such that starts from $P$, goes around the black hole less than 180° of angle, and then reaches $V$. We categorize such a ray as the "primary light ray", whose trajectory is illustrated as the solid line in Fig. 3. In this case, the incident angle $\psi$ of the light ray to the view point $V$ can be uniquely determined by the following three parameters:

1. the radial coordinate value $^3 v \in [a + \epsilon, R_{\max})$ at $P$,
2. the angle $\phi \in [0, \pi]$ enclosed by $P, O, \text{and} V$, and
3. the radial coordinate value $\psi \in [a + \epsilon, R_{\max})$ at $V$,  

where $[a + \epsilon, R_{\max}]$ specifies the range at which we can arrange the view point and polygon vertices. We set $\epsilon = 0.01a$ and $R_{\max} = 500a$ here. Once we obtain the angle $\psi$, the projected point $S$ on the perspective screen can be easily geometrically calculated. Let us categorize another type of ray such that starts from $P$, goes around the black hole more than or equal to 180° and less than 360° of angle, and reaches $V$, as the "secondary light ray", whose trajectory is illustrated as the dashed line in Fig. 3. Its incident angle $\psi'$ can also be uniquely determined by the given triplet $(p, \phi, v)$. Theoretically, we can also consider other types of rays that go around the black hole more than 360° of angle. However we discard them because the incident angles of such rays are contained within a quite narrow range and therefore the images of CG objects rendered with this kind of light rays are negligibly small.

As a consequence, if the points $P, O, \text{and} V$ are given,

1. we first determine the plane that contains all the three points,
2. calculate the three parameters $p, \phi, \text{and} v,
3. obtain the incident angles $\psi$ and $\psi'$ from the triplet $(p, \phi, v)$, and
4. calculate the points $S$ and $S'$ on the perspective screen.

Because $\psi$ and $\psi'$ obtained from $(p, \phi, v)$ can be recognized as the three argument functions $\psi(p, \phi, v)$ and $\psi'(p, \phi, v)$, respectively, we can expect to use the hardware trilinear interpolation unit on GPU with 3D textures to obtain the function $\psi$. Therefore this technique of utilizing the trilinear interpolation is not applicable when more than three parameters are required to calculate $\psi$ (or $\psi'$). Hence we note that our technique in this section is restricted for the spherically symmetric black hole and not applicable for black holes having more degrees of freedom. For example, the research paper [7] roughly estimates the potential of the interpolation for fast ray tracing in the case of the axi-symmetric black hole [2] and concludes that it is ineffective.

3.2 Mesh Structures for Perspective Projection

A 3D texture can be used not only for representing volumetric data in a CG scene but also for representing a function of three input parameters if the function becomes smoother as the interpolation error becomes smaller. Although the default mesh structure of the 3D texture is a cube of size 1 (=$(0, 1)^3$), we can change the structure by apply coordinate transformations so as to decrease the interpolation error. The aim of this section is to obtain an optimal mesh structure suitable for the functions $\psi(p, \phi, v)$ and $\psi'(p, \phi, v)$ discussed in the previous subsection.

Before proceeding to searching for the optimal mesh structure, let us estimate the permissible upper limit of the interpolation error $\Delta \psi$ of the incident angle. Suppose that the horizontal pixel resolution of the CG images is 2,000 and that the horizontal angle of view of the virtual camera is $40°$. Thus, letting $\Delta p$ be the error of the pixel position on a CG image, the error of the angle is derived as follows.

$$\Delta \psi = \frac{40°}{2,000} \Delta p = 0.02° \Delta p$$

If we give the condition $\Delta p < 1$, that is, the error is on the sub-pixel scale, it is implied that $\Delta \psi < 0.02°$. This is the maximum permissible error of the incident angle.

The simplest mesh structure is identified with the structure of the triplet $(p, \phi, v)$ itself, of which the two dimensional substructure $(p, \phi)$ is regarded as the black-hole-centric mesh structure illustrated in Fig. 4. This structure however is not acceptable for the primary rays because the interpolation errors around the view point are beyond the above permissible upper limit no matter how many mesh points are added to the mesh, that is, the mesh boundary does not match with the view point. In the similar reason, the view-point-centric mesh structure illustrated in Fig. 5 is also unacceptable due to the errors near around the black hole.

To resolve these problems, we introduce a new mesh structure $(s, t, u) \in [0, 1]^3$ that matches with both the boundaries of the view point and the black hole. Figure 6 illustrates its sub-mesh $(s, t)$, which consists of two kinds of circular arcs. One is the arc whose circular center is located on the line along the center of the black hole and the view point (see the arcs with parameter $s = 0.1, 0.2, \ldots, 0.9$ in Fig. 6). The other is the arc that passes through both the center of the black hole and the view point (see the arcs with parameter $t = 0.1, 0.2, \ldots, 0.9$ in Fig. 6). Mesh points are the crossover points of these two kinds of arcs. The calculation cost of the coordinate transformation $(p, \phi, v) \mapsto (s, t, u)$ is not very large because it requires at most 40 arithmetic operations, three square-root operations, and one arctangent operation (the details are omitted due to the limited paper space). Next, in
order to accommodate further smaller errors, one more coordinate transformation from \((s, t, u)\) to a new mesh \((s', t', u')\) is introduced. Figure 7 illustrates such an example of the optimal sub-mesh \((s', t')\) exhaustively obtained by the method explained in the next subsection. As a result, we can construct the optimized mesh for the primary rays by the two successive coordinate transformations: \((p, \phi, v) \mapsto (s, t, u) \mapsto (s', t', u')\).

Different from the primary light ray, we can use the black-hole-centric mesh \((p, \phi, v)\) for the secondary ray (see Fig. 4) and there is no need for concern about the boundary matching with the view point since the secondary ray goes around the black hole at least 180° so that any polygon point is regarded as sufficiently far from the view point even if they are geometrically close in the coordinate system. In order to normalize the mesh domain and accommodate smaller errors, the mesh \((p, \phi, v)\) is transformed to a new mesh coordinate \((s'', t'', u'')\) in \([0, 1]^3\).

3.3 Exhaustive Search of the Optimal Mesh

According to OpenGL, a 3D texture has the size of \(2^i \times 2^j \times 2^k\) \((i, j, k > 1)\) and the corresponding memory size is its multiple by 16. For example, the setting \(i = j = 8\) and \(k = 7\) requires 128 megabytes of memory on GPU.

Given a memory size \(M\), the author has exhaustively searched the optimal mesh structure (such as Fig. 7) for the primary rays as follows.

1. Initialize the sizes (triplet of integers) \((i, j, k)\) of the 3D texture such as \(16 \times 2^{i+j+k} = M\).
2. Initialize the shape of the mapping \((s, t, u) \mapsto (s', t', u')\) in some way.
3. Calculate \(\psi\) for each of \(2^{i+j+k}\) crossover points on the mesh.

(4) Check the error \(\Delta \psi = |\psi - \tilde{\psi}|\) between the true angle \(\psi\) and the linearly interpolated angle \(\tilde{\psi}\) at each of four million test points chosen different from the crossover points on the mesh, and summarize the maximum error \(\Delta \psi_{\text{max}}\), the average error \(\overline{\Delta \psi}\), and the average plus the three standard deviation \(\overline{\Delta \psi} + 3\sigma\).

(5) Modify the shape of the mapping \((s, t, u) \mapsto (s', t', u')\) in some way (with the idea of the so-called hill-climbing but almost randomly) and goto to Step (3).

(6) Determine the optimal mesh such that it gives the minimum \(\overline{\Delta \psi} + 3\sigma\) for the texture sizes \((i, j, k)\).

(7) Modify the texture sizes \((i, j, k)\) under the restriction \(16 \times 2^{i+j+k} = M\) and goto to Step (2).

(8) Determine the couple of the texture sizes \((i, j, k)\) and the optimal mesh such that it gives the minimum \(\overline{\Delta \psi} + 3\sigma\) for the given memory size \(M\).

In the same way, the optimal mesh for the secondary rays is determined.

Table 1 shows the experimental results of the interpolation errors for the primary rays in terms of the GPU memory usage. We see that \(\overline{\Delta \psi} + 3\sigma\) is under the maximum permissible error 0.02° when the texture memory size is 512 megabytes\(^6\). Thus we can statistically estimate that 99.7%\(^7\) of the interpolation errors are on the sub-pixel scale. On the other hand, \(\Delta \psi_{\text{max}}\) indicates that the errors are far bigger than the permissible error at some test points\(^8\). However, it does not seriously affect our research because other kinds of experiments we omit to give here show that such test points exist near at \(v \approx R_{\text{max}}\). Hence we can prevent them by putting the view point not far from the black hole. If we intend to put the view point far from the black hole, that is, we render super-telephoto images, we should construct a new mesh structure optimized for the super-telephoto condition, in which case we expect that we can use the black-hole-centric mesh in Fig. 4 instead of the complex mesh in Fig. 6 because no polygon point overlaps with the view point.

Table 2 shows the results for the secondary rays; the errors in this table are small enough even if we use a 3D texture of 16 megabytes.

It took about 116 hours and 114 hours to search for the optimal

---

\(^{a5}\) The author applied the bisection method of iteratively solving the corresponding IVP.

\(^{a6}\) In this case, \(i = j = 9\) and \(k = 7\). The number of mesh points is \(2^{i+j+k} = 2^{25}\), which is about thirty million.

\(^{a7}\) The actual ratio the author counted from the experimental results is 99.5%.

\(^{a8}\) The maximum error 4.014 in the case of 128 megabytes in Table 1 means that the searching procedure falls into the local minimum in this case because we focus on only the value of \(\overline{\Delta \psi} + 3\sigma\). As a matter of fact the second best solution gives \(\Delta \psi_{\text{max}} = 0.609\), \(\overline{\Delta \psi} = 0.006\), and \(\overline{\Delta \psi} + 3\sigma = 0.056\).
an example that renders a star field having tens of thousands of stars*9 on the celestial sphere. The view point exists near the black hole and watches the direction of the black hole. The black blank circular area at the center of Fig. 8 is considered to be the visual appearance of the black hole because no light ray from the black hole can reach the view point. Big, bright stars apart from the black hole are rendered with the primary rays, while small, dark ones near around the black hole are rendered with the secondary rays.

Note that the visual solid angle $d\Omega'$ of a star in the black hole spacetime differs from the corresponding solid angle $d\Omega$ in the flat spacetime due to the bending of light rays, and that the star becomes brighter than usual if $d\Omega' > d\Omega$ and vice versa. Hence we must prepare the ratio $d\Omega'/d\Omega$ at the same time as we prepare the quadruplet $(\psi', \gamma, L, f)$ in the preprocessing stage. Various shapes of stars including the cross shapes of brighter stars are implemented in the same way as in the book [21].

4.2 Rendering Lines

Lines can also be used in our rendering system. The difference from the usual OpenGL usage is that a straight line near around the black hole is not projected to a straight line on the perspective screen in general due to the light ray bending.

Figure 9 (a) illustrates an example of inaccurate projection of a straight line. Suppose that the primary light rays perspective project the two end points $P_1$ and $P_2$ of the line to the points $S_1'$ and $S_2'$ on the perspective screen, respectively. The usual rasterizing algorithm draws the straight line $S_1'S_2'$ while the accurately projected shape of the line could be the dashed, curved one in the figure. Similarly, the secondary light rays project $P_1$ and $P_2$ to $S'_1$ and $S'_2$, respectively, and the straight line $S'_1S'_2$ is drawn, which is also inaccurate in general.

In order to rasterize the accurate shape, we apply a recursive subdivision method [17] to line rendering. Figure 9 (b) illustrates the first level subdivision; the midpoint $P_m$ between $P_1$ and $P_2$ is projected to the point $S_m'$ by a primary ray, and the two straight lines $S_1'S_m'$ and $S_m'S_2'$ are drawn. Similarly, the same $P_m$ is projected to $S'_m$ by a secondary ray, and the two straight lines $S'_1'S_m'$ and $S_m'S'_2'$ are drawn.

*9 To make natural, realistic CG image, the positions and colors (color temperatures precisely) of the stars are obtained from Yale Bright Star Catalog [24].
and \( S'_m \) are drawn. Figure 9 (b) shows a better approximation than Fig. 9 (a). The condition for terminating the recursive subdivision is whether the pixel distance \( \delta \) (see Fig. 10) between the projected point \( S_m \) and the mid point \( S'_m \) between the projected points \( S_1 \) and \( S_2 \) is smaller than a certain threshold \( \Delta \), that is, \( \delta < \Delta \). We set \( \Delta = 1 \) in our implementation so that the approximation error of the subdivision can hardly be recognized by human eyes. Figure 9 (c) illustrates the case that the subdividing does not terminate for the primary rays but terminates for the secondary rays. The author implemented this dynamic, recursive line subdividing algorithm on the geometry shader (see Fig. 1) of a GPU. The recursion must also be terminated if the number of lines recursively generated is beyond the hardware upper limit of the geometry shader’s output buffer.

For example, Fig. 11 is the wireframe image of the Utah teapot of astronomical size; though the situation is quite unusual. The ring shape of the distorted image is called the Einstein ring [2]. It is known that the sun in our solar system has a gravitational radius of approximately 3.0 km if the sun became a black hole [2]. Under this assumption, the view point in Fig. 11 is at \( v = 30.0 \) km, the center of the teapot is at \( p_0 = 15.0 \) km, and its size \( s_0 = 6.0 \) km. All the lengths and sizes in this spacetime are calculated proportional to the gravitational radius \( a \).

Figure 12 is another CG example of rendering the \( x-y \) equatorial mesh lines. Note that the lines near the black hole are slightly red-shifted due to the gravitational potential. Figure 13 is an enlarged part of a CG image example in the extreme case that the shader’s output buffer is overflown. When the view point is almost in contact with the \( x-y \) equatorial plane under the similar situation to Fig. 12, the geometry shader generates too many lines beyond the output buffer’s limit because the light rays bend so strongly. Each disconnected line in Fig. 13 is the corresponding evidence. Resolving the overflow in such an extreme case is our next future work (see Section 5).

4.3 Rendering Triangular Polygons

Polygon rendering has essentially the same problem as line rendering; a triangle defined by three straight lines in the black hole spacetime is not projected to a triangle defined by straight
CG Example of a recursively subdivided (tessellated) triangular polygon near a black hole. The horizontal angle of view is 30°. The view point is at \( v = 50a \), the center of the polygon is at \( p_o = 200a \), and its size \( s_a = 200a \).

![CG Example](image1)

Fig. 14

CG image of the earth rotating a black hole. The horizontal angle of view is 17°. The view point is at \( v = 30a \), the center of the earth is at \( p_o = 200a \), and its radius \( s_o = 5a \).

![CG Image](image2)

Fig. 15

CG images of a US space shuttle rotating a black hole. The horizontal angle of view is 38°. The view point is at \( v = 10a \), the center of the polygon is at \( p_o = 5a \), and its size \( s_a = 2a \).

![CG Images](image3)

Fig. 16

CG image of a star field, a spiral galaxy, an accretion disk, and a space train of nine cars near a black hole. The horizontal angle of view is 60°. The view point is at \( v = 10a \), the center of the train is at \( p_o = 5a \), its length is 18a, and its width is 0.2a.

![CG Image](image4)

Fig. 17

4.4 Rendering Performance

Table 3 summarizes the performances of frames per second (fps) and primitives per second (pps) for ten kinds of CG scenes

<table>
<thead>
<tr>
<th>Scene Description</th>
<th>fps</th>
<th>pps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Star Field</td>
<td>120</td>
<td>300</td>
</tr>
<tr>
<td>Spiral Galaxy</td>
<td>60</td>
<td>150</td>
</tr>
<tr>
<td>Accretion Disk</td>
<td>30</td>
<td>75</td>
</tr>
<tr>
<td>Space Train</td>
<td>15</td>
<td>37.5</td>
</tr>
</tbody>
</table>

The term ‘tessellation’ is often used in the context of CG images to describe the process of dividing a polygon into smaller triangles. The paper discusses the recursive triangle subdivision method, which naturally extends from the line subdivision method. It notes that this method, when applied to polygons, can be thought of as a tessellation. At the initial step, we project the three vertices \( P_1, P_2, \) and \( P_3 \) of a given triangle to the points \( S_1, S_2, \) and \( S_3 \) on the perspective screen, respectively, and check exactly the same terminating condition in Fig. 10 for each of the three lines \( S_1S_2, S_2S_3, \) and \( S_3S_1 \). If all the three conditions hold, we rasterize the triangle \( S_1S_2S_3 \) without subdivision. Otherwise we subdivide the triangle to smaller triangles recursively, where we adopt the algorithm in Ref. [18] as the concrete subdivision algorithm. For example, Fig. 14 is the CG image of the actual rendering result of one triangular polygon. To show the triangles clearly, only their boundaries are painted with white color. The right bigger triangular mesh and the left smaller triangular mesh are rendered with the primary and secondary light rays, respectively. We see that the triangles nearer the black hole are smaller because the light rays bend stronger so that the recursion proceeds more deeply.

Three more CG examples are given in Fig. 15, Fig. 16, and Fig. 17. When rendering these images, the length \( L \) of every light trajectory mentioned in Section 3.4 is used for hidden surface removal. The so-called Z value cannot be used because some light rays move around the black hole more than 180°. Figure 15 is the CG image when the earth goes around the black hole. The earth is constructed from ten thousand triangular polygons onto which the texture image of the earth is mapped. We can see the similar CG image of a sphere, which is not defined by polygons, in the research paper [8]. The right hand side image of Fig. 16 is the CG image when a US space shuttle goes around the black hole, where the original modeling data [25] is shown in the left hand side. To the author’s best knowledge, no existing research has ever rendered such a complex shape of CG object (modeling data) constructed with more than ten thousand polygons in a curved spacetime. The last CG example given in Fig. 17 is the combined image of the following four CG objects.

1. A star field is rendered in the same way as in Fig. 8 as the background image.
2. A spiral galaxy texture image [10] mapped onto polygons is rendered as the background image.
3. An accretion disk [3], [27] is an astronomical disk structure formed by circumstellar dust that rotates spirally around the black hole. Because the disk is rotating fast in semi-light speed [11], the apparent color [12] of the disk changes reddish or bluish due to the Doppler shift. The similar CG image can be seen in many research papers (e.g., Ref. [13]).
4. A space train [13] of nine cars is rendered, where the original modeling data [25] is the Japanese train named Sakura.

*10 The original galaxy texture is an illustration by Mr. Shigemi Numazawa [26].
*11 Rendering fast moving CG objects violates our assumption (recall Section 2.2). However our CG program can render it accurately with a small program modification because the disk structure is invariant to time shift even though it moves fast.
*12 The color of the disk is calculated according to the theory of accretion disk [3] under some assumptions.
*13 The author pays homage to the Japanese manga “Galaxy express 999” written and drawn by Mr. Leiji Matsumoto.
rendered on two kinds of computers, PC1 and PC2. PC1 is a slightly outdated Desktop PC, Mac Pro (Mid 2012, OS X 10.10) with ATI Radeon HD 5770, and PC2 is a high-end notebook PC, MacBook Pro (Mid 2014, OS X 10.10) with NVIDIA GT 750M. The size of all the CG images is 1,280×720 pixels. Each of the experimental results given in the table is the average value when the view point moves slowly around the black hole, watching the direction of the black hole. The host program on CPU and the shader programs on GPU are written in OpenGL/GLSL.

We observe the following from the table.

Several pps values are beyond one million pps in PC1 and near one million pps in PC2, especially for the scenes with more than ten thousand primitives. The rendering performance of point sprites is considerably higher than those of lines and polygons because the primitive subdivision is not necessary for point sprites. In other words, the subdivision process is a serious bottleneck in the whole rendering process for lines and polygons. The pps values are almost proportional to the number of CG primitives for Scenes No. 2 to 7 in Table 3, because the rendering pipeline of the GPU is accelerated by the number of primitives. The rendering performance of Scene No. 8 on PC1 is extraordinary lower than the other cases; the reason is not understood yet. The reason why the rendering performances of PC1 and PC2 are not so different is understandable from several GPU benchmarks (e.g., Ref. [28]).

5. Conclusion and Future Work

In this paper we have developed the rasterization framework for the 3D CG in a spherically symmetric black hole spacetime based on Einstein’s theory of general relativity. The rasterization in this curved spacetime needs two key techniques. One is the trilinear interpolation for fast perspective projection and the other is the recursive subdivision of CG primitives for accurate rendering. As the result, we have achieved polygon rendering rates of about one million pps.

The following problems still remain on this research.

It is known that the recursive subdivision (tessellation) on the geometry shader is problematic from the view point of execution speed and GPU buffer overflow [19]. As mentioned in Section 4.4, the main reason of the slow rendering performance of lines and polygons is due to bottleneck of the tessellation. Hence our next work is to implement it with the tessellation specific shaders (the tessellation control shader and tessellation evaluation shader in OpenGL/GLSL [15], [21], [29] and the hull shader and domain shader in DirectX 11 [16]).

This paper assumes that any object stands at rest or moves slowly enough. However, the strong gravity attracts the object if it does not have an extremely powerful thruster. When the motion speed of the attracted object reaches near to light speed, the Doppler shift of light and other relativistic effects may be observed. In this sense the CG images shown in this paper include fades. The basic treatment of relativistically fast moving objects in ray tracing is given in [8]. We must develop its extended technique in rasterization, mainly extending the boundary value problem (BVP) and its solver. Let \((t'(\tau), r'(\tau), \theta'(\tau), \phi'(\tau))\) be a four-dimensional trajectory of a polygon point with an arbitrary parameter \(\tau\), where the trajectory of an object that moves freely under the influence of gravity is defined by the geodesic equation according to the theory of general relativity. Alternatively, the trajectory of an object that has a rocket thruster may be defined programmably, in which case we should precheck if the speed of the object is below the light speed, by calculating the magnitude of the line element \(ds\) along the programmed trajectory. The extended BVP must obtain the light ray trajectory \((t(\lambda), r(\lambda), \theta(\lambda), \phi(\lambda))\) such that the light ray crosses the view point at a certain time \(t(\lambda_0)\) and satisfies the equations \(t'(\tau) = r(\lambda_1), r'(\tau) = r(\lambda_1), \theta'(\tau) = \theta(\lambda_1), \phi'(\tau) = \phi(\lambda_1)\) with \(\lambda_0 > \lambda_1\). Here the technical problem is how to effectively solve it. The same tessellation techniques as in Section 4 are expected to be applicable to the solutions of this extended BVP.

The rendering performance of tablet computers is rapidly increasing. Table 3 shows that even a notebook PC can make CG images in realtime when the number of rendered polygons is not so excessive (not beyond ten thousand in the case of PC2). We expect that our research in this paper can be ported to tablet computers and run smoothly in realtime. It will have a large impact on education and entertainment.

\[\text{References}\]


\[\text{Table 3 Average frames per second (fps) and primitives per second (pps) of CG image generation.}\]

<table>
<thead>
<tr>
<th>Scene #</th>
<th>Primitives</th>
<th>CG image (Fig. #)</th>
<th># objects</th>
<th>PC1</th>
<th>PC2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Point sprites</td>
<td>Star field (Fig. 8)</td>
<td>9,110</td>
<td>456.0</td>
<td>4.15</td>
</tr>
<tr>
<td>2</td>
<td>Lines</td>
<td>Utah teapot (Fig. 11)</td>
<td>928</td>
<td>217.5</td>
<td>0.20</td>
</tr>
<tr>
<td>3</td>
<td>Polygons</td>
<td>X-y lattice lines (Fig. 12)</td>
<td>1,300</td>
<td>135.3</td>
<td>0.18</td>
</tr>
<tr>
<td>4</td>
<td>Utah teapot (no ref. image)</td>
<td>3,136</td>
<td>130.6</td>
<td>0.41</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Earth (Fig. 15)</td>
<td>9,800</td>
<td>83.9</td>
<td>0.82</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Galaxy (part of Fig. 17)</td>
<td>20,000</td>
<td>61.0</td>
<td>1.22</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Accretion disk (part of Fig. 17)</td>
<td>23,200</td>
<td>51.8</td>
<td>1.20</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Space shuttle (Fig. 16 (b))</td>
<td>27,096</td>
<td>13.5</td>
<td>0.37</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Space train (part of Fig. 17)</td>
<td>429,876</td>
<td>2.6</td>
<td>1.12</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Points sprites and polygons</td>
<td>Star field, galaxy, accretion disk, and space train (Fig. 17)</td>
<td>482,186</td>
<td>2.3</td>
<td>1.11</td>
</tr>
</tbody>
</table>

PC1: Mac Pro (Mid 2012) with ATI Radeon HD 5770
PC2: MacBook Pro (Mid 2014) with NVIDIA GT 750M

\[\text{Exploring Black Holes: Introduction}\]