Jinwook Kim · Soojae Kim · Heedong Ko · Demetri Terzopoulos

Fast GPU Computation of the Mass Properties of a General Shape and its Application to Buoyancy Simulation

Abstract To simulate solid dynamics, we must compute the mass, the center of mass, and the products of inertia about the axes of the body of interest. These mass property computations must be continuously repeated for certain simulations with rigid bodies or as the shape of the body changes. We introduce a GPU-friendly algorithm to approximate the mass properties for an arbitrarily shaped body. Our algorithm converts the necessary volume integrals into surface integrals on a projected plane. It then maps the plane into a framebuffer in order to perform the surface integrals rapidly on the GPU. To deal with non-convex shapes, we use a depth-peeling algorithm. Our approach is image-based; hence, it is not restricted by the mathematical or geometric representation of the body, which means that it can efficiently compute the mass properties of any object that can be rendered on the graphics hardware. We compare the speed and accuracy of our algorithm with an analytic algorithm, and demonstrate it in a hydrostatic buoyancy simulation for real-time applications, such as interactive games.

Keywords General-purpose computation on GPUs · Mass property computation · Physics-based animation · Rigid-body dynamics · Buoyancy simulation

J. Kim, S. Kim, H. Ko
Imaging Media Research Center
Korea Institute of Science and Technology
39-1 Hawolgok-dong, Seongbuk-gu, Seoul, 136-791, Korea
E-mail: {jwkim, lono, ko}@imrc.kist.re.kr

D. Terzopoulos
Department of Computer Science
University of California
Los Angeles, CA 90095, USA
E-mail: dt@cs.ucla.edu

1 Introduction

The fast calculation of mass properties, including the mass, center of mass, and products of inertia, is necessary for the dynamic simulation of solids. In rigid body dynamics, the mass properties are usually assumed to be constant during the simulation. Therefore, the computation can be performed in an initialization step and the computed values are used in the subsequent simulation. Hence, the computational cost to calculate mass properties is often negligible. In certain important cases, however, the mass properties can change during the simulation and complex geometric shapes may require expensive mass property computations.

Among these cases is the simulation of hydrostatic buoyancy. Buoyancy is a natural phenomenon resulting from the interplay between a fluid system and a floating rigid body system. If we assume a hydrostatic pressure condition for the fluid system, then we can simulate the motion of the rigid body floating in the fluid by applying a buoyant force to the center of mass of the instantaneous submerged volume, which is known as the center of buoyancy. The buoyant force itself is proportional to the instantaneous submerged volume. A problem here, of course, is that the submerged volume changes continuously. Consequently, the computation of its mass properties can be a major bottleneck of the simulation.

Most of the research in computing the mass properties of solid shapes can be applied only to specific solid representation schemes and, therefore, it may involve an expensive representation conversion process [11]. Gonzalez et al. [6] combined a polynomial free-form surface representation with the Gauss divergence theorem to efficiently calculate the moments of the enclosed object. However, their approach allows only piecewise polynomial surface patches. Mirtich [13] proposed an efficient method to compute the center of mass and higher-order moments for polyhedral objects. The proposed algorithm is based upon a three step reduction of the volume integrals to successively simpler integrals. The final step of the algorithm computes the required integrals over a

face from the coordinates of the projected vertices. This means that the computation is done by algebraic operations with vertex coordinate values. Even though this method is computationally efficient for fixed polyhedral objects, its efficiency can suffer if the geometric structure changes frequently as it may require an expensive reconstruction of a set of vertices and faces. Unfortunately, the typical situation in buoyancy simulations requires repeated updates of vertex coordinates and even of the number of relevant vertices. This is because the submerged volume is defined as the intersection of a geometric object representing the fluid system with a geometric object representing the floating rigid body.

In this paper, we propose a GPU-friendly algorithm to compute the mass properties determined by general geometries. Our approach is essentially image-based. Because of this, it is not restricted by the mathematical or geometric representation of rigid bodies. Regardless of the geometric representations employed, whether they be polyhedral approximation, free-form surfaces, constructive solid geometry, etc., if it is possible to render an object of interest on the GPU, then our algorithm can approximate the object's mass properties, exploiting the efficiency of the GPU.

Recent advances in the programmability of graphics hardware have enabled its use for general purpose computation, not restricted to rendering [16]. Various problems in scientific computation, including fluid dynamic simulation, the solution of linear systems of algebraic equations, nonlinear optimization, and volume rendering, have been addressed by taking advantage of the parallelism and programmability of GPUs [1,7–9,14, 17]. Moreover, programmable GPUs are getting faster and cheaper. Our algorithm accrues these benefits by exploiting the GPU to calculate mass properties. It first computes the mass, the center of mass, and the products of inertia by reducing volume integrals into surface integrals. It projects surfaces of the rigid body onto a plane that corresponds to the frame buffer of a rendering process. Next, it computes the integrands on the GPU. Finally, it performs a summation operation using a buffer reduction to obtain the desired result.

To perform the required integral operations over all the surfaces representing the non-convex geometric object, we use a depth-peeling algorithm to obtain each of the surface patches regardless of convexity. The depth-peeling is a fragment level depth sorting algorithm, which achieves a correct rendering of transparent objects that are located order independently [4,12]. The objective of the method is to find the fragments of geometry in a systematic manner. We focus our attention on this method because it can access all the fragments representing the geometry regardless of its convexity. We modify the original depth-peeling algorithm to obtain surface peels, which are surface patches beneath the fluid in our buoyancy simulation, as well as the intersection surface between the fluid and the rigid body.

The remainder of the paper is organized as follows: Section 2 reviews rigid body mass properties and derives them in the form of surface integrals over the projected plane. Section 3 introduces our GPU-friendly algorithm for computing the mass properties determined by non-convex geometry. Section 4 presents an error and performance analysis of our approach compared to the analytic method proposed by Mirtich [13]. Section 5 modifies an original depth-peeling algorithm to deal with hydrostatic buoyancy simulation and shows an example of interactive rigid body dynamics simulation under buoyancy. Finally, Section 6 draws conclusions from our work.

2 Rigid body mass properties

2.1 Computing mass properties with volume integrals

The *mass* of a rigid body is given by

$$m = \int_V \rho(x, y, z)\, dV, \tag{1}$$

where $\rho(x, y, z)$ is the *mass distribution* function of the body and V is its *volume*. If we assume $\rho(x, y, z)$ to be constant over the volume, the expression for the mass simplifies to $m = \rho V$. In this paper, the mass distribution function will be considered a constant value for simplicity.

The *center of mass* $\mathbf{r}$ and the *inertia tensor* $\mathbf{I}$ are given by

$$\begin{aligned} \mathbf{r} &= \frac{1}{V}\int_V \begin{bmatrix} x \\ y \\ z \end{bmatrix} dV, \\ \mathbf{I} &= \rho \int_V \begin{bmatrix} (y^2+z^2) & -xy & -xz \\ -yx & (z^2+x^2) & -yz \\ -zx & -zy & (x^2+y^2) \end{bmatrix} dV. \end{aligned} \tag{2}$$

2.2 Reduction to surface integrals on a projected plane

To calculate the mass properties of a rigid body efficiently, we exploit the *divergence theorem* as suggested by Gonzalez et al. [6]. According to the divergence theorem, an integral over the three-dimensional volume can be transformed into an integral over its boundary surface as follows:

$$\int_V \nabla \cdot \mathbf{f}\; dV = \int_{\partial V} \mathbf{f} \cdot \mathbf{n}\; dA, \tag{3}$$

where $\mathbf{f}$ is a continuously differentiable vector field defined on a neighborhood of V, where $\mathbf{n} = [n_x, n_y, n_z]'$ denotes the exterior normal vector of V along its boundary ∂V, and where dA is the infinitesimal surface area of the boundary. When the volume is represented by a bounding polyhedron, its boundary is the set of planar

polygons comprising its faces. If we set $\mathbf{f} = [0, 0, z]'$, then we obtain the volume as $V = \int_{\partial V} z n_z \, dA$. Similarly, setting $\mathbf{f}$ in turn to $[0, 0, xz]'$, $[0, 0, yz]'$, and $[0, 0, \frac{1}{2}z^2]'$ yields $\int_V x \, dV = \int_{\partial V} xz n_z \, dA$, $\int_V y \, dV = \int_{\partial V} yz n_z \, dA$, and $\int_V z \, dV = \int_{\partial V} \frac{1}{2} z^2 n_z \, dA$, respectively.

Now, we slightly modify (3) by projecting the boundary surface area element dA onto the xy plane. From Figure 1, we see that the relationship between the infinitesimal surface area dA and the projected surface area $dx\,dy$ is $dx\,dy = |n_z| \, dA$ if the surface normal vector has unit length.

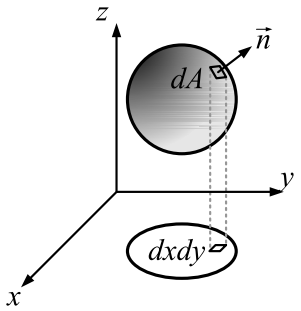


Fig. 1 Projection of the infinitesimal surface area element.

Finally, we obtain the volume V and the center of mass $\mathbf{r} = [r_x, r_y, r_z]'$ as follows:

$$\begin{aligned} V &= \int_{\partial V} \mathrm{sgn}(n_z) z \, dx\, dy, \\ r_x &= \frac{1}{V} \int_{\partial V} \mathrm{sgn}(n_z) xz \, dx\, dy, \\ r_y &= \frac{1}{V} \int_{\partial V} \mathrm{sgn}(n_z) yz \, dx\, dy, \\ r_z &= \frac{1}{2V} \int_{\partial V} \mathrm{sgn}(n_z) z^2 \, dx\, dy, \end{aligned} \tag{4}$$

where $\mathrm{sgn}(x)$ denotes the signum function which extracts the sign of a real number x. Note that the integrals are computed on the planar surface area, which is achieved by projecting the surface boundary onto the xy plane. When the surface area element dA is projected on the xy plane, it will be singular if $n_z = 0$. Hence, an improper choice of $\mathbf{f}$ (e.g., $\mathbf{f} = [x, 0, 0]'$ to compute the volume) can lead to a singularity at the boundary of a projected surface, where it would require division by a very small number. Our proposed $\mathbf{f}$s, however, only require multiplication by $\mathrm{sgn}(n_z)$, thus avoiding the singularity problem at the boundaries.

The inertia tensor $\mathbf{I}$ is

$$\mathbf{I} = \rho \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{bmatrix}, \tag{5}$$

where the *moments and products of inertia* are similarly given as follows:

$$\begin{aligned} I_{xx} &= \int_{\partial V} \mathrm{sgn}(n_z) x^2 z \, dx\, dy, \\ I_{xy} &= \int_{\partial V} \mathrm{sgn}(n_z) xyz \, dx\, dy, \\ I_{yy} &= \int_{\partial V} \mathrm{sgn}(n_z) y^2 z \, dx\, dy, \\ I_{xz} &= \frac{1}{2} \int_{\partial V} \mathrm{sgn}(n_z) xz^2 \, dx\, dy, \\ I_{yz} &= \frac{1}{2} \int_{\partial V} \mathrm{sgn}(n_z) yz^2 \, dx\, dy, \\ I_{zz} &= \frac{1}{3} \int_{\partial V} \mathrm{sgn}(n_z) z^3 \, dx\, dy. \end{aligned} \tag{6}$$

3 Computing mass properties on the GPU

3.1 Shader implementation

The programmability of recent graphics hardware and the various choices of precision and formats of framebuffers enable us to implement mass property computations on GPUs in an easy and flexible way. The integrands in equations (4) and (6) can be evaluated discretely at each pixel in a framebuffer by GPU programming. The process is straightforward:

1. Render the geometry with an orthographic projection onto the xy plane;
2. Evaluate the integrands on a fragment shader;
3. Encode the evaluated values at the output buffers.

The number of parameters that must be computed is 10 in total, including 1 for volume, 3 for the center of mass, and 6 for the moments and products of inertia. To store these parameters, we use three framebuffers, each of which can contain four values in the red, green, blue and alpha channel. This can be efficiently implemented using the "multiple render target" capability of recent graphics hardware, which enables the fragment shader to save per-pixel data in multiple buffers.

Hence, we obtain color buffers containing the values of integrands in equations (4) and (6). Furthermore, the integration of the values over the projected plane area can be performed by reading back fragment color values of the framebuffers and summing them up, or by using a buffer reduction algorithm as will be explained in the next section. A fragment shader can be implemented in the OpenGL Shading Language [15] very easily, as follows:

```
// homogeneous coordinate of a point on the surface
varying vec4 p;

// z component of the surface normal
varying float n_z;

void main(void)
{
  float c = sign(n_z) * p.z;

  // (rx, ry, rz, V)
  gl_FragData[0] = c * p;

  // (Ixx, Ixy, Ixz, .)
  gl_FragData[1] = p.x * gl_FragData[0];

  // (Iyy, Iyz, Izz, .)
  gl_FragData[2] = c * vec4(p.y * p.y, p.y * p.z, p.z * p.z, 0);
}
```

Note that the fourth components of `gl_FragData[1]` and `gl_FragData[2]` are not used.

A potential problem is how to generate color buffers covering all the surface fragments of the geometric shape. Consider the case of a sphere. The surface of a sphere can be divided into two patches—the north and south hemispheres—according to the direction of surface normals. If we look at the sphere from the negative z viewing direction, the line of sight will intersect the sphere twice. That is, the typical rendering pipelines will render two fragments from those two surface patches on one pixel in the framebuffer and, therefore, the resulting color buffer will contain only one of the fragments from the two surface patches regardless of the choice of the depth test function. To resolve this problem, we use the depth-peeling algorithm discussed in the next section.

3.2 Depth-peeling

Using the standard depth test function of the 3D graphics API, we can obtain the nearest surface fragment from the eye at each pixel. Although the second nearest or other fragments may be required in some areas, there is no straightforward way to obtain the nth nearest fragment. One possible solution is to use a depth-peeling algorithm, which is a fragment-level depth sorting technique [12]. Depth-peeling can be implemented as a multipass algorithm. In the first rendering pass, the geometries are rendered using a normal "less-than" depth function. This will yield a depth buffer containing the depth values of the nearest surface of the geometry. In the next rendering pass, only the fragment for which depth is greater than the depth values in the buffer from the previous pass are rendered. Then the depth buffer will contain the depth values of the next nearest surface of the geometry, and so on. The process repeats until the depth values of all the surface fragments are found. The depth-peeling technique introduced by Everitt [4] requires a shadow buffer to peel away the surfaces by comparing depth values. However, since recent GPUs and APIs support "render-to-texture" capabilities and the direct manipulation of pixel values on fragment processors using shading languages, depth-peeling can be implemented using programmable GPUs and the modification of the algorithm is even easier.

For our objective of computing mass properties, we can apply the standard depth-peeling algorithm with the shader developed in the previous section. As a result, we obtain n textures containing the enumerated integrands in equations (4) and (6), where n is a total number of peels.

3.3 Two-dimensional integrals over the projected area using buffer reduction

Using the textures obtained in the previous section, we compute the two-dimensional integrals over the projected surfaces in order to obtain mass properties. A straightforward way to perform the integration is to read all evaluated integrands from framebuffers and sum them. Given current graphic memory interfaces, however, reading back a texture memory directly into system memory can yield significant latency. To tackle this problem, we use buffer reduction [2]. To summarize the buffer reduction technique, a fragment program reads two or more values from the buffer and computes a new value using the reduction operator, which in our case is an addition operation. These passes continue until the output is reduced to a single value, the sum. In general, this process takes $O(\log n)$ passes, where n is the number of elements to reduce. Figure 2 illustrates a reduction operation to calculate the sum.

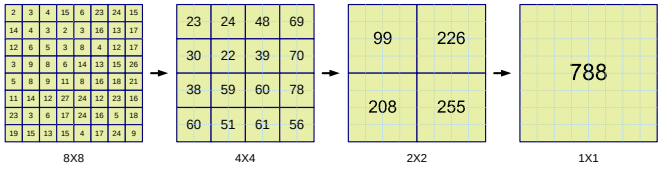


Fig. 2 Summation reduction procedure.

4 Performance

In this section, we compare our algorithm in terms of accuracy and speed with the analytic method developed by Mirtich [13]. Since the analytic method used in this test is restricted only to polyhedra, we use shapes approximated by polyhedra as test objects. Figure 3 illustrates some of these objects. It is important to note, however, that our algorithm can be applied to any model that can be rendered on graphics hardware. All the tests were run on a 2.53GHz Pentium 4 CPU with an NVIDIA GeForce 7800 GTX GPU. 32-bit floating point textures were used for framebuffers. Table 1 lists all the geometric test objects and the number of peels for each test object.

Fig. 3 Some polyhedral test objects.

object	vertices	faces	peels
cube	8	12	2
sphere	422	840	2
teapot	821	1628	6
torus	1024	1922	4
knot	1440	2880	8
bunny	2557	5110	6
pipe	4626	9252	6

Table 1 Geometric information for the test objects.

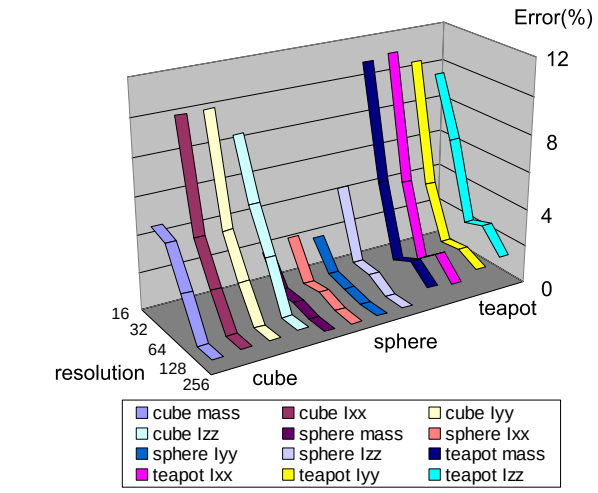
4.1 Error analysis

We measured the relative error of the mass and the moments of inertia I_{xx}, I_{yy}, and I_{zz} at various framebuffer resolutions. The other mass property values are very small for our test objects, because the shapes are approximately symmetric along axes. Our approach computes integrands for each fragment on the GPU, where we use texture memories as framebuffers. Hence, the resolutions of the framebuffers are critical for accurate results. As shown in Figure 4, a resolution of 32×32 was sufficient to compute the mass properties within a 5% error bound.

4.2 Performance analysis

We now compare the performance of our algorithm and the analytic method. If we assume that the cost of vertex processing on the GPU is negligible compared to the cost of fragment processing, the complexity of our algorithm is approximately $O(kn^2)$, where k is the number of rendering passes for the depth-peeling and n is the framebuffer resolution along its width or height. On the other hand, the complexity of the analytic method is $O(m)$, where m represents the number of faces of the polyhedron. Figure 5 shows a comparison of the computation times for the analytic method and our GPU-based method at three different framebuffer resolutions.

We observed that our GPU-based approach is comparable to the analytic method in terms of computational cost. At 64×64 resolution, our algorithm outperforms the analytic method for moderately complex shapes such as the bunny or the pipe. However it also shows the downside of quadratic complexity for a resolution of 128×128 or more. For example, it is obvious that the analytic method is preferable to our GPU-based method for low-polygon-count models such as a cube. The computational



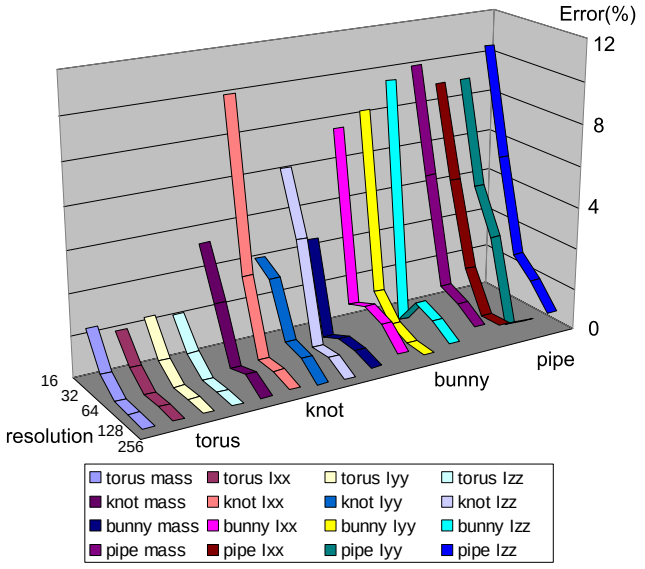


Fig. 4 Relative error comparisons.

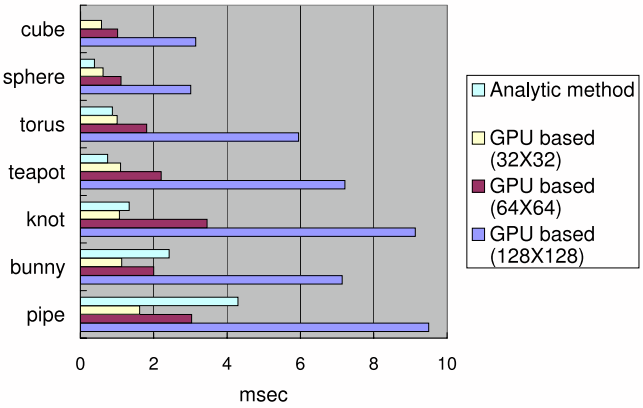


Fig. 5 Computational time comparison of analytic method and GPU-based method.

cost of the analytic method for the cube is so small that we could not distinguishably display it in the graph.

5 Case study: Buoyancy simulation

The beauty of our image-based approach is that it is not restricted to any particular mathematical or geometric shape representation. It can efficiently compute the mass properties of arbitrary objects, so long as they can be rendered efficiently on graphics hardware. As an example application of our algorithm, we will now demonstrate an interactive, hydrostatic buoyancy simulation.

5.1 Hydrostatic buoyancy

One of the most popular simplifications of fluid motion is the *shallow water model* [10] which assumes zero viscosity and considers only two-dimensional motions. An interesting fact of the shallow water model is that the pressure field is characterized by the *hydrostatic equilibrium condition*:

$$p = \rho g h, \tag{7}$$

where g is the gravitational acceleration and h is the depth of the fluid. This very simple pressure model works well with the shallow water model, and it corresponds exactly to the observation of Archimedes.

According to *Archimedes' principle*, a body immersed in a fluid experiences a vertical *buoyant force* equal to the weight of the fluid that it displaces. The buoyant force acts on the center of mass of the submerged volume. Figure 6 illustrates a rigid body partially immersed in a fluid. Assuming a stationary fluid system, two forces are acting on the body at this instant. The first is the force of gravity that acts downwards at the center of gravity C, while the second is a buoyant force which acts upwards at the *center of buoyancy* B, which is the center of mass of the immersed part of the rigid body (assuming that the immersed portion consisted of fluid). The magnitude of the buoyant force is proportional to the weight of the submerged volume of fluid. The imbalance between gravity and the buoyant force induces a torque that will rotate the body to restore a static equilibrium.

The simulation of fluid motion is out of the scope of our work.[1] Instead we focus on the rigid body motion of an object floating on fluid due to the hydrostatic buoyant force. To simulate hydrostatic buoyancy, we compute the volume and the center of mass of the submerged part

[1] Foster and Metaxas [5] demonstrate a simplified scheme for coupling buoyant objects to the results of a Navier-Stokes fluid simulation. Carlson et al. [3] simulate the interplay between rigid bodies and viscous incompressible fluid.

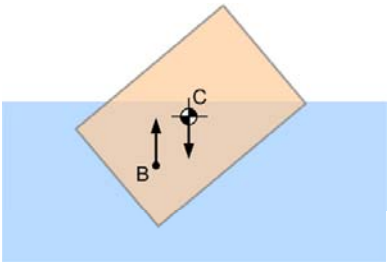


Fig. 6 Buoyant force and gravity acting on a partially submerged rigid body.

of the body at every simulation time instant. If the geometries of a fluid body and a rigid body are complicated, calculating their intersection requires a considerable amount of computation and can become a bottleneck in the simulation process. In the following section, we tackle this problem by modifying the depth-peeling algorithm.

5.2 Boundary surfaces of an intersection volume of a non-convex geometry and a fluid surface

We improve the original depth-peeling technique to account for all the projected fragments of the geometry below the fluid surface. For simplicity, let us assume that the signs of the z components of the fluid surface normal vectors do not change. Our algorithm considers surfaces from the rigid body and the fluid surface intersecting the geometry separately. The multi-pass rendering procedure to handle the surfaces of a submerged volume is as follows (note that an orthographic projection is applied to render the scene with the negative z viewing direction as shown in Figure 7):

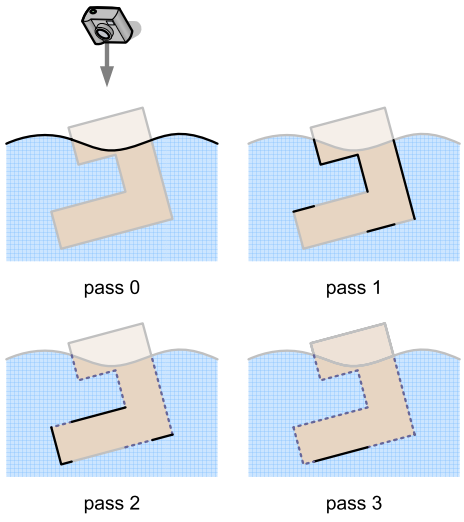


Fig. 7 Multi-pass rendering to obtain all surface patches.

Pass 0: Render the surface of the fluid, storing its depth values into a texture T_{dw} as a reference.

Pass 1: Render the object geometry to a texture T_1. In our fragment shader, the integrands in equation (4)—i.e., $\mathrm{sgn}(n_z)xz$, $\mathrm{sgn}(n_z)yz$, $\mathrm{sgn}(n_z)z^2$, and $\mathrm{sgn}(n_z)z$—are evaluated and the output color is composed of their values. Also, the depth values are stored in a texture T_{d1}. During this rendering pass, the fragments whose depth value is less than the fluid surface depth are discarded to inhibit the operation. The texture T_{dw} generated in rendering Pass 0, is used to lookup the depth value of the fluid surface. In Figure 7, the solid black lines correspond to the fragments.

Pass 2: Render the geometry to a texture T_2. As in the previous rendering pass, the integrands are evaluated and their values are assigned to the output color. Here, only the fragments whose depth value is greater than the fluid surface depth *and* the depth value of T_{d1} are accepted in order to peel away the surface patch obtained in the previous rendering pass. In Figure 7, the dashed lines indicate peeled away fragments. A depth texture T_{d2} is initialized with T_{d1} and overwritten with the depth values of the currently processed fragments.

Pass n: Repeat the same process as in rendering Pass 2 until all the object fragments are found and evaluated.

Thus, we obtain n textures, and the texture T_n contains the evaluated integrands of the nth surface patch.

Now, the only remaining surface patch is the fluid surface intersecting with the rigid body geometry. As illustrated in Figure 8, the surface patches of rendering Pass 1 consist of upward and downward faces. The fluid surface intersecting with the rigid body geometry can be obtained by drawing the fluid surface only for those fragments having a downward normal in rendering Pass 1. Note that a more efficient implementation results if the fragment shader can write a stencil bit into the output framebuffer in rendering Pass 1. Finally, we evaluate the integrand for the fluid surface patch intersecting the rigid body geometry and write the value in a texture T_w.

In summary, our algorithm requires a total of $n + 2$ rendering passes to cover all the surface patches of a partially submerged rigid body geometry, where n represents the maximum number of intersection points of the submerged part of the geometry with the z axis. The first rendering pass generates a reference depth texture from the fluid surface. In the next n rendering passes, integrands are evaluated for each fragment of the geometry surface and the resulting values are stored in textures T_i. The final rendering pass evaluates the integrand for the fluid surface patch that intersects the rigid body geometry and stores the values in a texture T_w.

Finally, we apply the summation reduction procedure described in Section 3.3 to evaluate the integral expressions for the volume of the immersed portion of the object and the center of buoyancy in order to evaluate the buoyant force and its point of application in the object.

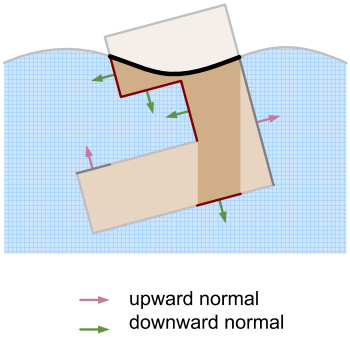


Fig. 8 Intersecting the surface of the fluid body with a rigid body.

5.3 Simulation example

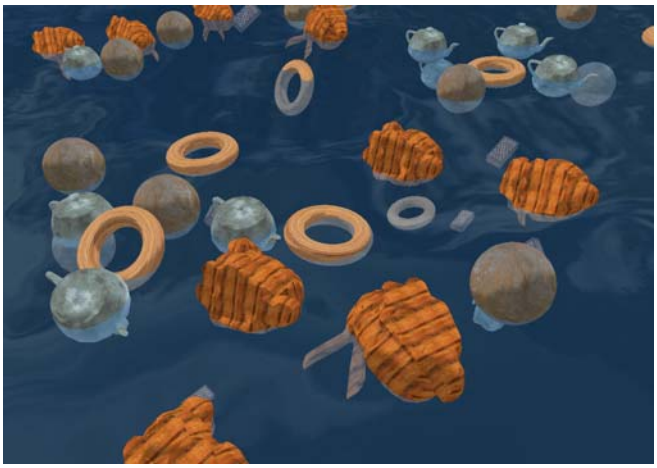


Fig. 9 Interactive simulation of 50 rigid bodies floating in water.

Figure 9 shows a typical scene from our interactive simulations of buoyant objects. Ten spheres, 10 rectangular boxes, 10 tori, 10 teapots, and 10 Stanford bunnies were tested. Boxes with density higher than that of the water were observed to sink as expected. We also modeled a viscous drag force acting at the center of buoyancy with magnitude proportional to the submerged volume and the square of the body velocity. The simulation runs on the CPU of a 2.53GHz Pentium 4 PC employing an NVIDIA GeForce 7800 GTX GPU. The average frame rate of the example shown in the figure was 16 frames/sec. Over 90% of the computational resources were consumed in calculating the buoyant force.

For spherical and rectangular bodies, depth-peeling was applied twice to compute the submerged volume of the object geometries. For the teapot and Stanford

bunny bodies, depth-peeling was applied a maximum of 6 times, but in most cases 3 or 4 peels sufficed to cover the submerged volume. The framebuffer resolution used in this example was 32×32, allowing at most 5% approximation error. The leftmost images in Figure 10 show the gravity force (downward blue arrow) and buoyant force (upward yellow arrow) acting on a bunny, a torus, and a teapot. The remaining images are color buffers that encode the integrands for each peel, as described in the previous section. Since the framebuffers use a floating point texture format that cannot be illustrated properly, we have transformed the values so that they map to a color range of [0,1].

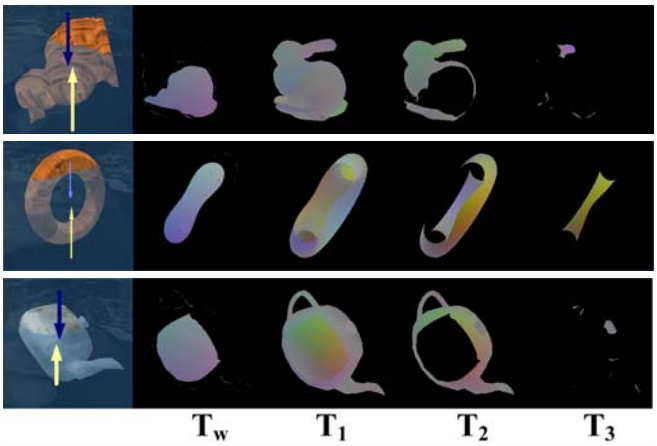


Fig. 10 Color encoded integrands of each peel for the buoyant bunny, torus, and teapot (left).

6 Conclusion

We have proposed a GPU-friendly algorithm for computing the mass properties of a rigid body represented by a general geometry. We formulated the mass properties as surface integrals on a projected plane, avoiding singularities at the boundaries. We also showed that depth-peeling techniques can be exploited to tackle non-convex geometries. Our approach is essentially image-based. Consequently, it can efficiently compute mass properties as long as the geometries can be rendered using graphics hardware.

We applied our algorithm to simulate rigid body motion in a real-time hydrostatic buoyancy simulation. The mass properties of the submerged volume were efficiently computed without an explicit reconstruction of the intersecting geometry between the fluid and the rigid bodies. Our algorithm approximates mass properties fairly accurately, even using low resolution framebuffers. Our interactive simulation demonstrates that the proposed algorithm can be applied to animate floating rigid bodies on a stationary fluid system in a fast and plausible way.

Acknowledgements The material presented herein is based upon work supported by the Information and Telecommunication National Scholarship Program supervised by IITA and the Ministry of Information and Communication, Republic of Korea.

References

1. Bolz, J., Farmer, I., Grinspun, E., Schröoder, P.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid. ACM Trans. Graph. **22**(3), 917–924 (2003)
2. Buck, I., Purcell, T.: GPU Gems 2, chap. A toolkit for computation on GPUs. Addison-Wesley (2004)
3. Carlson, M., Mucha, P.J., Turk, G.: Rigid fluid: Animating the interplay between rigid bodies and fluid. ACM Transactions on Graphics **23**(3), 377–384 (2004)
4. Everitt, C.: Interactive order-independent transparency (2001). URL citeseer.ist.psu.edu/everitt01interactive.html
5. Foster, N., Metaxas, D.: Realistic animation of liquids. Graphical Models and Image Processing **58**(5), 471–483 (1996)
6. Gonzalez-Ochoa, C., McCammon, S., Peters, J.: Computing moments of objects enclosed by piecewise polynomial surfaces. ACM Transactions on Graphics **17**, 143–157 (1998)
7. Hillesland, K.E., Molinov, S., Grzeszczuk, R.: Nonlinear optimization framework for image-based modeling on programmable graphics hardware. ACM Transactions on Graphics **22**(3), 925–934 (2003)
8. Krüger, J., Westermann, R.: Acceleration techniques for GPU-based volume rendering. In: VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03), p. 38. IEEE Computer Society, Washington, DC, USA (2003)
9. Krüger, J., Westermann, R.: Linear algebra operators for GPU implementation of numerical algorithms. ACM Transactions on Graphics (TOG) **22**(3), 908–916 (2003)
10. Layton, A.T., van de Panne, M.: A numerically efficient and stable algorithm for animating water waves. The Visual Computer **18**(1), 41–53 (2002)
11. Lee, Y.T., Requicha, A.A.: Algorithms for computing the volume and other integral properties of solids. I. Known methods and open issues. Communications of the ACM **25**(9), 635–641 (1982)
12. Mammen, A.: Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. IEEE Computer Graphics and Applications **9**(4), 43–55 (1989)
13. Mirtich, B.: Fast and accurate computation of polyhedral mass properties. Journal of Graphics Tools **1**(2), 31–50 (1996)
14. Moreland, K., Angel, E.: The FFT on a GPU. In: HWWS '03: Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware, pp. 112–119. Eurographics Association, Aire-la-Ville, Switzerland (2003)
15. Rost, R.J.: OpenGL Shading Language. Addison-Wesley Longman, Redwood City, CA (2004)
16. Thompson, C.J., Hahn, S., Oskin, M.: Using modern graphics architectures for general purpose computing: A framework and analysis. In: Proceedings of the ACM/IEEE International Symposium on Microarchitecture, pp. 306–317 (2002)
17. Wu, E., Liu, Y., Liu, X.: An improved study of real-time fluid simulation on GPU. Computer Animation and Virtual Worlds **15**(3–4), 139–146 (2004)