Exploding Object via Particle System

Papon Lapate
paponl@mit.edu

Namo Samuthrsindh
paramuth@mit.edu

1 Problem Description and Motivation

Explosions serve as a crucial visual element with diverse applications, notably in genres like science fiction movies. For a computer-generated explosion to appear more genuine, the objects within the scene must be affected and scattered due to the explosion's force. Thus, we aim to discover a method for computing and illustrating these exploding objects, enabling us to capture the intricate interactions and forces involved in the object's explosion. If we possess a method to depict an exploding object that can capture numerous intricate forces, we can employ it in various scenarios. For instance, we could simulate an exploding object within a scene featuring multiple sources of explosions, where not all of them occur simultaneously. Even in situations that don't directly involve an explosion, this technique can be applied. For example, an exploding object can be utilized in a collision scene, simulating the impact of the collision and the dispersion of objects.

2 Background

In OpenGL, there exists an optional shader positioned between the vertex shader and the fragment shader, known as the geometry shader [1]. This shader receives a collection of vertices constituting a singular primitive, transforms these vertices, and subsequently transmits them to the next shader. The capability to perform transformations is what gives this shader the potential to contribute to the generation of an exploding object.

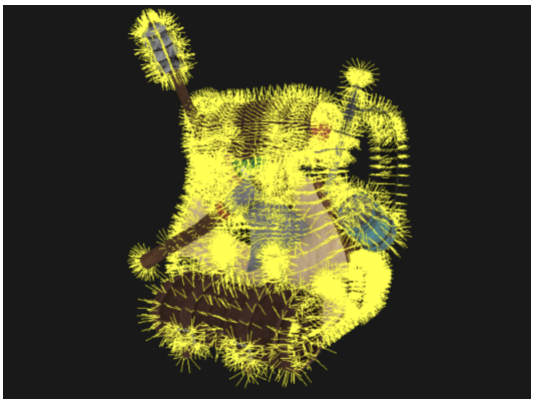


Figure 1: Normals (depicted in yellow) of each face indicate the direction in which each face is displaced by the geometry shader [4].

A prevalent method for generating an explosion through a geometry shader involves displacing each vertex along the normal of the face (a type of primitive) it belongs to [3]. This is achievable because the geometry shader calculates the face's normal using the information received about the set of vertices forming the face. The outcome is an object undergoing an explosion, with each face dispersing outward, imparting the sensation of an explosion emanating from the center of the mesh.

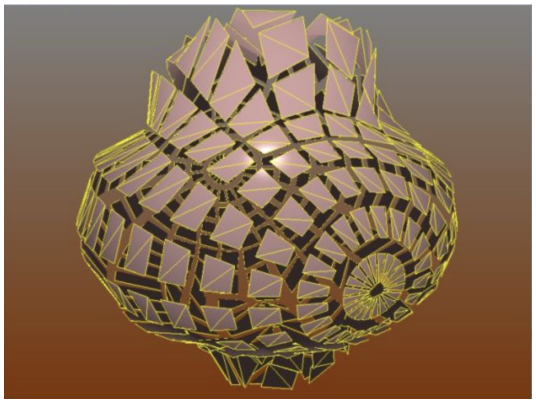


Figure 2: Exploding sphere rendered with the geometry shader, with velocities in each direction scaled by varying factors [3].

However, after delving into the concepts behind the geometry shader, we recognize that this approach comes with certain limitations when it comes to representing intricate particle systems within an explosion. One issue is that the existing implementations uniformly handle primitives, making it challenging to customize the effect for a more intricate outcome where each component of the object undergoes unique transformations. While it might be conceivable to establish an additional system within the geometry shader to calculate distinct transformations for each primitive, this approach is inconvenient and susceptible to errors. The difficulty lies in the necessity to distinguish and individually process each primitive when the geometry shader only receives input as a set of vertices forming a primitive.

To overcome these constraints, we opt for an alternative method to generate an exploding object, specifically by employing a particle system. A particle system is a technique employed to characterize the motion of objects, represented by "particles," over time, detailing the state of all particles at a specific moment. It serves to articulate the alterations in the position and velocity of each particle influenced by diverse forces. Utilizing a particle system allows us to retain all information essential for computing the explosion within the system. Additionally, we can consolidate all computations within an integrator. By configuring the states of the particles, we gain the ability to differentiate between each primitive based on how the particle states are established.

3 Approach

We divide our implementation into 3 steps: subdividing the triangle mesh, creating the particle system for imitating an explosion from a point, and using the said system to simulate collision with a simple object.

3.1 Mesh Subdivision

Intuitively, when an object explodes, we would expect the object to be broken into several small pieces. If we implement explosion by simply moving each triangle in the mesh, then the explosion may seem unrealistic if the triangles are too large.

The implementation by Toapanta solves this issue by breaking down each triangle into several points (which are distributed evenly across the triangle) and spreading them away from the centroid of the triangle [2]. We will use a similar method here. However, instead of dissolving the mesh into points, we'll subdivide the triangles in the mesh into smaller sub-triangles instead. More specifically, there will be n^2 sub-triangles (for some $n \in \mathbb{N}$), each of which is similar to the original triangle but with size length shrunk down by a factor of n. If we denote by $P(a, b)$ the points with barycentric coordinate $\left(\frac{a}{n}, \frac{b}{n}, \frac{n-a-b}{n}\right)$ with respect to the triangle, then the sub-triangles consist of

- $\frac{n(n+1)}{2}$ sub-triangles with vertices $P(a, b)$, $P(a+1, b)$, and $P(a, b+1)$ for $a, b \geq 0$ such that $a + b < n$, and

- $\frac{n(n-1)}{2}$ sub-triangles with vertices $P(a,b)$, $P(a-1,b)$, and $P(a,b-1)$ for $a,b \geq 1$ such that $a+b \leq n$.

We precompute this subdivision before the explosion begins (but without moving them away from the original position yet), and then we can move each sub-triangle independently.

Note that finer subdivision means that there will be more triangles we need to render, which can be costly, so the size of sub-triangles must be adjusted on a case-by-case basis depending on the required rendering speed.

3.2 Explosion System

Now we need to figure out how to move the sub-triangles when an explosion happens. To do this, we use a method similar to physics-based simulation: we create a structure that contains the state (positions and velocities) of the sub-triangles and a system that contains information about how those states change due to the effect of the explosion. We then create an integrator which, for each time we render, uses the state (from the previous rendering time) and the system to calculate the new state at present.

For the integrator, we simply use a Runge-Kutta integrator. Now we'll describe the formula contained in the system.

The system stores the points that are the centers of the explosion. To calculate the acceleration of each sub-triangle due to an explosion at each center, we will calculate the effect of the explosion on the 3 vertices, then we average them and set the average as the acceleration of all vertices. This makes sure that each sub-triangle stays rigid as the movements of all of its vertices are the same.

Now, to calculate the acceleration at each vertex due to each explosion, we use a formula similar to point light attenuation: the acceleration is in the direction $\hat{r}$ of the (unit) ray from the center of the explosion to the vertex, with magnitude roughly proportional to $\frac{1}{d^2}$ where d is the distance between the center of the explosion and the vertex. We also add some linear and constant terms to the denominator to avoid the acceleration blowing up near the center of the explosion. Therefore, our acceleration will be of the form $\frac{1}{ad^2+bd+c}\hat{r}$ where a, b, c are parameters for the explosion.

There are also more parameters and additional features that we use to improve realism:

- For convenience, we have a parameter t for the time the explosion starts.
- Since an explosion is an "impulse" that doesn't stay forever, we have a parameter ϵ for each explosion, which determines the duration (after the explosion starts) when the explosion is "active," after which the explosion isn't taken into account when calculating acceleration.
- To ensure each triangle moves smoothly, we make it so that the acceleration from each explosion diminishes continuously as time passes (with the acceleration reaching 0 as the duration of the explosion reaches ϵ). More precisely, we multiply the acceleration by $1 - \frac{\delta}{\epsilon}$ where δ is the duration between the start of the explosion and the present.
- Finally, since the explosion "expands" from the center continuously, we have a parameter (expansion rate) p that describes how far the effect of the explosion can reach at a given time. More specifically, at time δ after the explosion starts, only the vertex at a distance no more than δp will be accelerated. (This can also be used to ensure not the entire mesh will move when the explosion is not very strong.)

For more than one explosion, we simply calculate the acceleration due to each bomb independently and add them together.

3.3 Simulating Collision

Now that we have a way to animate explosions from point sources, we can use it to implement collision with simple objects.

Imagine a scene where a heavy object, moving at high speed, hits a fragile object (e.g. a ceramic vase). We would expect to see the vase broken into many pieces, each of which moving away from the direction that heavy object comes from. Moreover, the pieces that start near the point of contact

should be affected the most by the impact, with the pieces far away getting less possibly not affected at all.

This is very similar to how we implement explosion from a point, which suggests that we can use the same calculation we use for explosion to animate collision as well.

Since calculating collision naively is rather expensive, we chose to test our collision idea by using a spherical ball for the heavy object. That way, we only need to use the ball's radius and origin to calculate collision with each triangle in the fragile object's mesh. (We'll discuss in more detail the problem with collision for more complex objects in the discussion section.)

Clearly, the time the "explosions" start should be the time when the two objects collide, and since the effect is the largest at the point of contact, that point should be the center of the explosion. ϵ is set to be a constant.

The rest of the parameters (a, b, and c used in the calculation of acceleration and expansion rate p) should depend on how "hard" the impact is. The speed of the ball is certainly a factor, but the direction in which the ball moves into the fragile object is also important — the ball crashing directly into the object definitely has more impact than if the ball merely grazes the side of the object. Therefore, we say that acceleration and p should depend on value similar to $|\vec{v} \cdot \hat{n}|$, where $\vec{v}$ is the velocity of the ball and $\hat{n}$ is the normal of the sub-triangle the ball hits. In practice, we decide to use $l := |\hat{v} \cdot \hat{n}|^k \times |\vec{v}|$ where $\hat{v}$ is the normalized version of v and k is a parameter (we separate $|\vec{v}|$ from the rest so that the acceleration won't blow up when $|\vec{v}|$ slightly changes if k is large). This means a, b, and c will be inversely proportional to l since they are inversely proportional to acceleration.

To enable all these, we add another function to the explosion system that we can use to add more explosions to the system by inputting l and the time and position of contact.

There is one issue: the actual movements of the objects are not continuous in practice and the ball may clip into the fragile object, which means that the intersection between the ball and a sub-triangle may not be a single point but a curve. We resolve this by using the point in the sub-triangle that is closest to the center of the ball as the center of the explosion, and we also push the sub-triangle in the direction of its normal (or opposite of that) until the sub-triangle is outside the ball to make sure that the explosion will push the sub-triangle away from the ball even further.

Calculating the point in the sub-triangle that's closest to the center of the ball can be done as follows:

1. First, we project the center of the ball onto the plane containing the sub-triangle. If the projection is inside the sub-triangle, then that projection is the closest point.
2. Else, the closest point must be on one of the sides of the triangle. We can then, for each of the 3 sides, find the point closest to the center of the ball and choose the closest of the three.
3. For each side of the sub-triangle, finding the point that is closest to the center of the ball can be done by, again, projecting the center of the ball onto the line containing that side. If the projection is on that side, then this must be the closest point. Else, the closest point to the center of the ball is one of the two endpoints of the side, whichever is closer to the center of the ball.

Finally, since collision is not supposed to be as "devastating" as an explosion, we may ignore the mesh subdivision step without sacrificing realism (as long as the triangles in the mesh are not too big.)

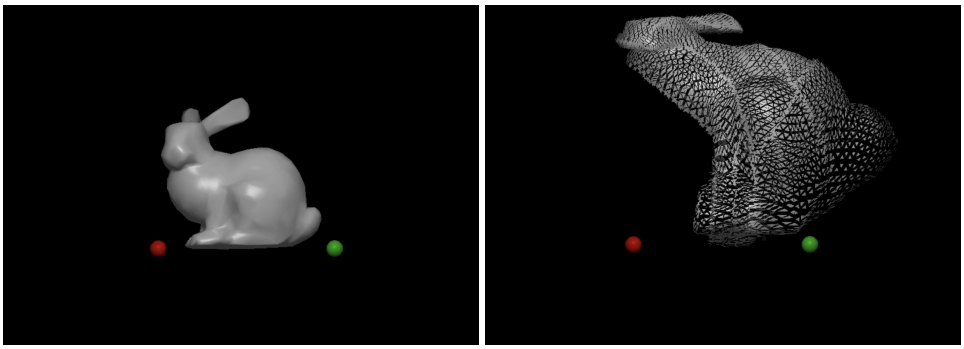
4 Results

Utilizing the particle system method, we simulate two situations involving the use of an exploding object: **Point(s) Explosion** and **Collision via Explosion**.

4.1 Point(s) Explosion

Point(s) Explosion directly involves the utilization of an exploding object. In this scenario, there are two explosion sources represented by a red and green sphere in Figure 3. The explosion initiates from the red sphere initially, and after a brief interval, the explosion from the green sphere commences. This

is manifested in the bunny mesh as it begins to move away from the red sphere before subsequently moving away from the green sphere. This underscores two advantages of utilizing a particle system. Firstly, there is no restriction to a single explosion source. Additionally, it provides the flexibility to adjust the timing differences for the initiation of each impact.



(a) (b)

Figure 3: Rabbit Mesh before (a) and after (b) an explosion

The complete live-captured animation of Point(s) Explosion can be found here. Before rendering the video, we decrease the velocities of individual vertices, and subsequently, we enhance the speed using an editing tool before uploading to YouTube. This adjustment is necessary due to our hardware limitations, which would otherwise result in latency during the rendering process.

4.2 Collision via Explosion

Collision via Explosion is a utilization of an exploding object that employs an explosion to represent the effect of a collision. In this case, the sphere maintains a constant velocity before colliding with the bunny mesh. To depict the collision, we introduce explosion sources around the points of contact between the sphere and the bunny mesh, detonating them simultaneously with the collision. The explosion of the bunny mesh effectively communicates the impact of the collision. We have the flexibility to adjust the velocities of individual vertices and determine the degree of mesh partitioning based on the desired strength of the impact we want to convey.

(a) (b)

Figure 4: Rabbit Mesh before (a) and after (b) a collision

The complete live-captured animation of Collision via Explosion can be found here.

5 Discussion

We employed a method to depict a detonating object through the utilization of a particle system. This approach enabled us to create an exploding object triggered by two distinct point sources that

detonate at separate times as well as a scattering object resulting from collisions. By configuring the particle system appropriately, our technique for rendering exploding or dispersing objects can be applied to accommodate any other scenarios necessitating the depiction of such objects.

Nevertheless, there are certain realistic physics aspects that we haven't considered in our simulation.

Firstly, we have not factored in the torque applied to the primitives. In other words, we haven't accounted for the rotational motion induced by unequal forces acting on each vertex of a primitive. To tackle this issue, we simplify the simulation by disregarding the spin and making the assumption that the acceleration on each vertex is consistent, averaging the effects—although this deviates from the actual dynamics observed in the real world. Considering the rotational motion would enable us to implement a more realistic representation of an exploding object. (Another physics-related problem is that we calculate the acceleration directly without considering the mass of each sub-triangle. However, this is a rather minor issue and can be resolved easily by, for example, assuming that the mass is proportional to the area of each sub-triangle.)

Another aspect where we employ unrealistic assumptions for simplicity is in the mesh division. In actual explosions, not all faces of the mesh are divided into an equal number of sub-meshes. Those closer to the point of explosion should be divided into a larger number of sub-meshes than those farther away. Furthermore, at an equivalent distance, faces with larger surface areas should be subdivided into a greater number of sub-meshes compared to those with smaller areas. In our simulation, we assume equal division of all mesh faces. For a more realistic simulation, it is necessary to calculate the explosion's impact strength at each radial distance from the center of the explosion, taking into account the face area. This calculation is then used to determine the suitable size for mesh division for that face.

Another type of issue with our simulation is those concerning how we deal with collision. One such issue is that of efficiency: as noted before, if we use more complex objects than a ball when we simulate collision, then we'd need to check collisions between all pairs of triangles from the moving object and from the fragile object, which is inefficient if implemented naively. Another minor problem is similar to what we found in our collision simulation: the moving object may clip into the fragile object, and the intersection between triangles will be a line segment instead of a curve. Since we don't have a "center" of the moving object, we can't use the exact same calculation for the general case anymore. However, the main idea should still be the same: we can assume that, for a short period of time, the moving object has constant velocity, then we can find the point where the moving object first hit the sub-triangle in the fragile object's mesh and set the center of explosion accordingly. (We can also consider moving the sub-triangle along the velocity of the moving object instead of its normal, which should also ease the calculation for the distance we should move the sub-triangle.)

References

[1] D. Shreiner, G. Sellers, J. M. Kessenich, and B. Licea-Kane, *OpenGL Programming Guide : The Official Guide to Learning OpenGL, Version 4.3*, 8th ed. Boston, MA, USA: Addison-Wesley, 2013. [Online]. Available: https://www.cs.utexas.edu/users/fussell/courses/cs354/handouts/Addison.Wesley.OpenGL.Programming.Guide.8th.Edition.Mar.2013.ISBN.0321773039.pdf

[2] G. J. T. Toapanta, "Exploding Bunny," GitHub, https://github.com/GiovannyJTT/exploding_bunny (accessed November 13, 2023).

[3] JeGX, "Mesh Exploder with Geometry Shaders," Geeks3D, http://www.geeks3d.com/20140625/mesh-exploder-with-geometry-shaders/?fbclid=I (accessed Nov. 13, 2023).

[4] J. De Vries, "Geometry Shader," Learn OpenGL, https://learnopengl.com/Advanced-OpenGL/Geometry-Shader (accessed Nov. 13, 2023).