Griffin Shea

Game Engine Acceleration Using Property-Centric Architecture and Compute Shaders

W.I.P.

Title Page

Abstract

Acknowledgements

Table of Contents

List of Figures

List of Tables

1. Introduction

It is best to leave the introduction to be done last.

2. Motivation and Objectives

Realistic or pseudo-realistic physics simulation has become an expected feature of most modern game engines, especially in the triple-A games industry. We would like for these game engines to be able to handle large numbers of physics objects or particles colliding and applying forces upon one another and themselves (such as the player object moved by internal forces applied through user input) fast enough for real time frame updates. Some key calculations involved in physics simulations such as the integration of linear and angular acceleration and velocity to calculate new positions and orientations for rigid-body physics objects are performed independently per-object. This means, we could benefit greatly by performing these calculations for all objects concurrently in a compute shader program ran on the GPU, rather than taking the standard approach of performing these calculations one after another, per object, on the CPU. This principle can also be applied to other systems and functions of game code; Specifically, wherever we see that a large number of like-objects need to be updated and these updates can be applied to each object independently of the other objects in the group, we can likely save processing time by loading the data relevant to those objects onto the GPU and update them all at once with a compute shader.

Code reusability and modularity are essential for modern game engine development. The earliest object-oriented game engines were designed using a rigid game-object class hierarchy to share common functionalities between game-object types through inheritance. In recent years however, component-based designs, which utilize the delegation pattern to share common functionality, have become preferred because they are relatively easier to develop with and eliminate the problems of diamond inheritance (or alternatively, duplication of code across branches of the class hierarchy). However, component-based game engine architecture tends to become spaghetti-like because game-object properties are stored within components which often need to be referenced or edited by other components contained by the same game-object or other objects entirely. Furthermore, component-based architectures are not very well optimized for loading data into the GPU to process game-object updates in compute shaders because there can be a significant processing overhead when collecting the properties of a game-object that are stored across several components. A better approach may be to store all game-object properties in a centrally located data container and use static code in the place of components to implement the functionality to update different kinds of game-objects – this is called a property-centric architecture. Property-centric architecture also offers a streamlined approach to loading game data into GPU memory by allowing us to simply collect a set of properties associated with objects of a certain type from the central game data container as requested by a compute shader.

The first major objective of this project is to investigate the usage of compute shaders to simultaneously update all instances of a game-object type on the GPU for the purpose of accelerating game engine performance. The second major objective of this project is to demonstrate the advantages of a property-centric architecture for the purpose of streamlining the incorporation of compute shaders into a game engine and to assess the strengths or weaknesses of the property-centric approach compared to hierarchical and component-based game engine designs. To accomplish these objectives, we will implement a rigid-body physics simulation within the game engines that we develop in order to comparatively assess the strengths of property-centric architecture and compute shaders for handling a large number of objects. Thus, a sizeable portion of this report will be dedicated to documenting our rigid-body physics simulation. This report will therefore double as a brief guide into that domain of real-time programming as well.

3. Methodology Overview

This project will begin with an existing game engine (produced by the author) and make a series of modifications to it in order to ultimately show improvements in game engine performance. The existing engine can be briefly described as a hybrid between component-based and hierarchical designs. This description will be expanded in a subsequent chapter, but for now it is only pertinent to know that the engine is limited to a few basic object types, a collision detection system, and a scene graph that is responsible for making calls to the renderer to draw each object. To ensure that the results of this project are going to be applicable to the actual development of real games, we believe that by completing an implementation of a rigid-body physics simulation and ragdolls, the existing engine will be in a better position to provide a meaningful performance benchmark from which to demonstrate the advantages of switching to a property-centric architecture and the use of compute shaders.

So, the first tasks of this project will be to implement rigid-body physics and ragdolls using constraint solvers and then design some demos using these new objects that we can run to record a performance benchmark. This version of the engine will be referred to as the Stage 1 prototype. After this, the engine will be edited to implement property-centric architecture: game-object properties will be extracted from the game-object classes and will instead be stored by the game in a central location within a series of nested containers. Then, the update code for each class in the game-object hierarchy will be reformatted into static functions which update each ‘object’ (in fact, these objects do not really exist, only the data that comprises them, which is associated with an object ID) of that type by editing data in the game data container. Finally, the Stage 2 prototype will be complete when compute shaders have been implemented within the static game-object functions. It will also be helpful, if not necessary, to create a new static class which will streamline the logic for loading data in and out of the GPU and executing compute shaders. The final tasks of this project will be to run the demos once again on the Stage 2 prototype and compare the observed performance results to those recorded after testing the Stage 1 prototype. In order to ensure the results accurately reflect the change in the engine’s architecture, care will be taken to avoid editing the actual algorithms involved in updating game-objects, where possible.

4,5,7.

- desc of existing engine’s main systems including collision detection work, plus its history/background

- relate its design to hierarchical, component-based, how it is hybrid, and show in what ways these cause problems

4. Established Approaches to Game Engine Design

Hierarchical

Component-based

Hybrid architecture

5. Stage 1 Prototype Architecture

Description of stage 1 engine (it’s a hybrid)

What work is pre-existing what was added

7. Collision Detection

Colliders – discrete collision detection

Collision tree – broad step: AABB, KD-tree

Collision tree – narrow step: GJKSM, EPA, contact point generation

8. Rigid-Body Physics

The rigid-body is a popular and useful representation for simulation of solid objects that are subject to forces and/or torques and the collisions between such objects. The rigid-body physics simulation implemented in the Stage 1 prototype mainly follows [Baltman and Radeztsky, 2004], which is our preferred source, although, [Bourg and Bywalec, 2013] was also helpful in the early stages of development.

8.1 Rigid-Body Definition

The main body of code governing rigid-body physics is naturally located in the RigidBody class, which defines a RigidBody object with the following properties: 3-component position vector (p), orientation quaternion (q), scalar mass (m), 3x3 inertia matrix (I), 3-component force vector list (shown below as the sum $\sum F_i$), 3-component torque vector list (shown below as the sum $\sum T_i$)[†], and three scalar coefficients for restitution (e), static friction (μ_s), and kinetic friction (μ_k), as well as a pointer to a Collider object (collider) and a pointer to a SceneNode object (sceneNode).

As we will see, the inertia matrix is essential for calculating how a rigid-body's angular velocity will change after a collision, among other things. This matrix is determined by the shape of the rigid-body, or rather, how the mass of an object is distributed around its center of mass (which is the position). The inertia matrix is therefore calculated by a call to the collider's getInertiaMat function (passing mass as a parameter), which is implemented for OriBox, Sphere, Cylinder, and Capsule.[‡] Also, it is calculated about the object's principal axis of rotation – this will become relevant later.

We use the Verlet integration method using the velocity-less equations (in reference to [Baltman and Radeztsky, 2004]) of motion so rather than keeping track of linear/angular velocity or momentum, we store two additional values representing the "last position" (p^*) and the "last orientation" (q^*) of the rigid-body and calculate velocities implicitly as needed for calculations.

8.2 Rigid-Body Basic Functions

Linear and angular velocities can be calculated with the following functions, respectively, where Δt is the time taken between render frames, which is stored by Renderer:

$$v(p, p^*) = \frac{p - p^*}{\Delta t}$$

$$\omega(q, q^*) = \frac{2 \cdot \mathrm{vec}\left(\ln\left(\mathrm{norm}(q \cdot \overline{q^*})\right)\right)}{\Delta t} \text{ [§]}$$

In the numerator of the function $\omega(q, q^*)$, $\mathrm{norm}(q \cdot \overline{q^*})$ represents calculating a 'difference' unit quaternion for a rotation between the current and previous orientation quaternions. The remaining operations in the numerator term $2 \cdot \mathrm{vec}\left(\ln\left(\mathrm{norm}(q \cdot \overline{q^*})\right)\right)$ represent transforming $\mathrm{norm}(q \cdot \overline{q^*})$ into a 3-component vector form appropriate to representing angular velocity.

[†] In this implementation, $\sum F_i$ and $\sum T_i$ are not actually stored as 3-component vector lists, but rather they are lists of callback functions that return a calculated value. This format is quite convenient for forces or torques that would be affected by distance or other factors, such as magnet. In the demos below however, the only force acting on bodies is the force of gravity, which is constant, and no torques are utilized.

[‡] The formulas for the first three shapes are in [Bourg and Bywalec, 2013] while the formula for the inertia matrix of a capsule was found online at [Lovrovic, 2015]. Formulas for the inertia matrix of other shapes can be found at various sources online – for example https://en.wikipedia.org/wiki/List_of_moments_of_inertia – or calculated by hand.

[§] The 'vec' function here creates a 3-component vector using the vector part of the given quaternion. For the logarithm of a unit quaternion: $\ln(\mathrm{quat}[\cos\alpha, n\sin\alpha]) = \mathrm{quat}[0, \alpha n]$.

$$M(q,I,a) = q * \left((\bar{q} * a)I^{-1}\right)^{**}$$

We also define the above function to calculate the 3-component moment of inertia vector of a rigid-body about an arbitrary axis. The axis vector a is multiplied by $\bar{q}$ to rotate it relative to the principal axis of rotation before being multiplied by I^{-1} and again rotated using q. This is necessary because I was calculated relative to the principal axis of rotation in the constructor. Alternatively, I could be recalculated each frame relative to the arbitrary current orientation of the rigid-body, but this solution is excessively expensive.

8.3 Rigid-Body Update

The Verlet integration method is applied to update the positions and orientations for all rigid-body objects once each frame. When the equations of motion below are executed in sequence, they give the algorithm used for this calculation.

$$f = 1 - \Delta t \cdot F$$

$$\Delta p = (p - p^*) \cdot f + \sum F_i \frac{\Delta t^2}{m}$$

$$p^* = p$$

$$p = p^* + \Delta p$$

$$\Delta q = 2 \cdot \mathrm{vec}\left(\ln\left(\mathrm{norm}(q \cdot \overline{q^*})\right)\right) \cdot f + M(q, I, \sum T_i)\Delta t^2$$

$$q^* = q$$

$$q = \mathrm{norm}\left(q^* + q^* \cdot \mathrm{quat}\left[0, \frac{\Delta q}{2}\right]\right)$$

Note that the same transformations seen in $\omega(q, q^*)$ are applied to calculate Δq as a 3-component vector which is later ‘added’ to q using the formula $q = \mathrm{norm}\left(q^* + q^* \cdot \mathrm{quat}\left[0, \frac{\Delta q}{2}\right]\right)$. Also, a form of constant fluid friction or drag over time has been applied to both linear and angular velocity by the implementation of the coefficient f. This prevents objects from carrying on moving or spinning *ad infinitum*, which is a behaviour we do not observe on Earth as air is constantly applying a drag on all objects that are engulfed by it. It also helps with stability as it ensures that the energy of the rigid-bodies decreases between frames and does not spiral out of control due to calculation errors resulting from the floating-point precision problem. The value of F is a user-specified constant, which we have set to a default value of 0.25.

8.4 Rigid-Body Collision

** $q * v$ is used as shorthand for the calculation $q \cdot v \cdot \bar{q}$ which represents a rotation of vector v by the unit quaternion q.

Our simulation also implements rigid-body collision resolution. The colliders attached to each rigid-body are added to the CollisionTree held by Level when all objects are initialized and added to the game. Each frame, after all game-objects have been updated, the CollisionTree detects all collisions between colliders. If both colliders are solid and at least one of them belongs to a rigid-body, the RigidBody `resolveCollisions` class function is called to determine whether one of the colliders is static and then solve the collision.

8.4.1 Collision Impulse

A rigid-body collision is solved by calculating a collision impulse and a friction impulse to apply to both rigid-bodies and then correcting the position of the two objects to prevent them from intersecting. When a rigid-body collides with a static object, the static object is treated as a rigid-body with infinite mass for the purposes of these calculations (although, we have 'hard-coded' a separate function to handle this behaviour).

$$r_1 = c_1 - p_1$$

$$v_{c_1} = v(p_1, p_1^*) + \omega(q_1, q_1^*) \times r_1$$

$$r_2 = c_2 - p_2$$

$$v_{c_2} = v(p_2, p_2^*) + \omega(q_2, q_2^*) \times r_1$$

$$v_r = v_{c_1} - v_{c_2}$$

$$J = \frac{-v_r \cdot \hat{n} \cdot \left(1 + \frac{e_1 + e_2}{2}\right)}{m_1^{-1} + m_2^{-1} + \hat{n} \cdot M(q_1, I_1, r_1 \times \hat{n}) \times r_1 + \hat{n} \cdot M(q_2, I_2, r_2 \times \hat{n}) \times r_2}$$

$$C = J\hat{n}$$

Above is the formulaic algorithm for calculating the collision impulse between a rigid-body 1 and rigid-body 2. There are three new terms $\hat{n}$, c_1, c_2 which are passed into `resolveCollisions` via the collision artefact generated by the `Collider.checkCollision` function: $\hat{n}$ is the normalized form of the separation vector n, which is the shortest distance to move two colliders out of intersection (pointing from body 2 to body 1); c_1 and c_2 are the collision contact points relative to each rigid-body collider involved in the collision – i.e., the point at which the two bodies are touching when they collide. It follows that v_{c_1} and v_{c_2} are the local velocities of each rigid-body at the contact point and v_r is the relative velocity of the whole system at the point of contact. J represents the magnitude of the collision impulse, which gives the collision impulse vector C when used to scale $\hat{n}$.

SECTION ON FRICTION TO BE COMPLETED

8.4.2 Applying Impulses

The impulse vector C is applied to each object in order to update the trajectories of the rigid-bodies after the collision.

$$s_1 = \frac{m_2}{m_1 + m_2}$$

$$q_1^* = \text{norm}(q_1 - q_1 \cdot \text{quat}\left[0, \frac{\left(\omega(q_1, q_1^*) + M(q_1, I_1, r_1 \times s_1 C)\right)\Delta t}{2}\right])$$

$$p_1^* = p_1 - \left(v(p_1, p_1^*) + \left(\frac{s_1 C}{m_1}\right)\right)\Delta t$$

$$s_2 = \frac{m_1}{m_1 + m_2}$$

$$q_2^* = \text{norm}(q_2 - q_2 \cdot \text{quat}\left[0, \frac{\left(\omega(q_2, q_2^*) + M(q_2, I_2, r_2 \times -s_2 C)\right)\Delta t}{2}\right])$$

$$p_2^* = p_2 - \left(v(p_2, p_2^*) + \left(\frac{-s_2 C}{m_2}\right)\right)\Delta t$$

First, the two s terms are calculated representing the relative mass of each rigid-body and is used to scale C to divide the impulse between the two objects. Then, the p^* and q^* values representing the 'last position' and 'last orientation' are edited to change the velocities of the rigid-body objects. This final step in collision resolution looks quite different from the sources because this implementation uses the velocity-less equations of motion, so linear and angular velocity can not be directly modified. The formulae above have been derived from the formulae below, where v_1 and v_2 are the associated linear velocities and ω_1 and ω_2 are the associated angular velocities rigid-body 1 and rigid-body 2:

$$v_1 = v_1 + \frac{s_1 C}{m_1}$$

$$\omega_1 = \omega_1 + M(q_1, I_1, r_1 \times s_1 C)$$

$$v_2 = v_2 + \frac{s_2 C}{m_2}$$

$$\omega_2 = \omega_2 + M(q_2, I_2, r_2 \times s_2 C)$$ [††]

9. Ragdolls and Constraints

With the rigid-body implementation discussed thus far, one can easily begin to simulate more complex and interesting phenomena by implementing a few basic constraints. A ragdoll is a kind of physics object that is commonly found in video games and typically used to animate pseudo-realistic death animations. The principles behind this effect can however be applied to simulate many less gruesome physical behaviours, such as a free-swinging door. A ragdoll is essentially a group of rigid-bodies connected to

[††] They are analogous to those given in [Baltman and Radeztsky, 2004], 7.

each other at joints which ensure that they do not fall apart and do not bend into impossible contortions. To do this, each joint object solves an angle constraint and a pin constraint once each frame after the game objects are updated and before the collision detection and resolution step. But before getting into constraints and how they work, it will be useful to outline the structure of our ragdoll object. [needs citations in this section]

9.1 Ragdoll Object

Our implementation includes three kinds of objects: Ragdoll, RigidBone, and RigidJoint. The ragdoll as a whole is comprised of a tree of RigidBone and RigidJoint objects and Ragdoll contains the root node, which is a RigidBone object representing the hips. Ragdoll also contains the instructions for assembling a human ragdoll including declarations for twelve capsule shaped RigidBody objects in its constructor. Each RigidBody is contained by a RigidBone, so we also have twelve RigidBones. In addition to its RigidBody, a RigidBone also contains a pointer to its parent (another RigidBone) plus a list of children, which are RigidJoint objects. In this case, one RigidBone represents one limb or part of a human body that is connected to one or many other limbs/body parts. For example, the hips ‘bone’ is connected to the waist ‘bone’ and the left and right thigh ‘bones.’

Each RigidJoint in a bone’s list of children connects that bone and its rigid-body to one other bone and its rigid-body by fixing their positions, orientations, and momentums together each frame so that they never appear separated and keep their relative orientations within set bounds. So, a RigidJoint contains both a parent and a child RigidBone pointer plus: two 3-component position offset vectors (o_1 and o_2, relating to the parent’s rigid-body – 1, and the child’s rigid-body – 2), one for the parent and one for the child, specifying a point relative to each rigid-body that will ‘pin’ the two together; a joint orientation quaternion (j) representing the default relative orientation between the two rigid-bodies; and finally, a 3-component angular freedom vector which determines how far the child bone can pitch (φ), yaw (θ), and roll (ψ) away from the default orientation relative to its parent. Ragdoll is setup to have eleven RigidJoints because there is one connecting to each rigid-body, except for the root, i.e., the hips, which is unconstrained.

We also disable collision detection for adjacent bones to allow for more creative freedom in the placement of the capsules that make up the body, for example, they overlap where the thighs meet the hips. Also, since the angle constraint will handle the work of making sure jointed bones do not rotate inside one another, allowing collision detection would be an unnecessary waste of processing power.

9.2 Constraints

Once each frame, after each game object (including the rigid-bodies) has been updated, the Ragdoll `constraintUpdate` function is called for each ragdoll, which traverses the tree and solves each joint’s angle and pin constraint. A constraint, with regard to physics programming, is essentially a condition or equation which we would like to keep true: for example, a distance constraint between two points a and b would take the following form, where l is the desired constant distance:

$$0 = |x - y| - l$$

Here, x and y could be the positions of two point particles which are tied together by an indestructible and inflexible bond of length l. Because the objects are updated separately at each step in the simulation, they may be subject to a variety of forces and impulses independently of one another and become separated or come closer together. To fix this problem, we modify each position by distributing the difference between the two (with regard to their relative point mass) so that the equation above holds true and the bodies again appear separated by the constant distance l. Additionally, to ensure conservation of momentum, we also need to calculate an impulse to fix the momentums of each particle coinciding with the modification of their positions. However, in this implementation, since we use the Verlet integration method with the velocity-less equations of motion, and momentum is calculated implicitly with regard to the current and last position, we can accomplish both tasks in one simple step, as will be shown below.[‡‡]

We use two constraints on each joint to keep the structure of the ragdoll coherent. The first is an angle constraint, which ensures that the relative orientation of each parent-child pair of bones stays within specified bounds (see below). Then, a pin constraint (highly related to the distance constraint) is used to keep the bones from falling apart.

9.3 Pin Constraint

The pin constraint defines a pair of relative positions offset from the center of each rigid-body which are constrained to occupy the same position in space:

$$t_1 = q_1 * o_1$$

$$t_2 = q_2 * o_2$$

$$d = (p_1 + t_1) - (p_2 + t_2)$$

$$0 = |d|$$

To solve this constraint, we calculate an impulse and apply it to each rigid-body's current position and orientation moving them so that the pin point on each body coincides. An impulse is used, rather than a simple translation, because, since the two bodies are pinned at one point only, they are free to rotate relative to each other about that point. This process is similar to the one used to resolve rigid-body collision though it is not identical:

$$\hat{d} = \text{norm}(d)$$

$$K = \frac{-1}{m_1^{-1} + m_2^{-1} + \hat{d} \cdot M(q_1, I_1, t_1 \times \hat{d}) \times t_1 + \hat{d} \cdot M(q_2, I_2, t_2 \times \hat{d}) \times t_2}$$

$$D = Kd$$

$$q_1 = \text{norm}(q_1 + q_1 \cdot \text{quat}\left[0, \frac{M(q_1, I_1, t_1 \times D)}{2}\right])$$

[‡‡] This is, in fact, the reason why we chose this method and these equations.

$$p_1 = p_1 + \frac{D}{m_1}$$

$$q_2 = \text{norm}(q_2 + q_2 \cdot \text{quat}\left[0, \frac{M(q_2, I_2, t_2 \times -D)}{2}\right])$$

$$p_2 = p_2 + \frac{-D}{m_2}$$

By modifying p and q via impulse vector D, we are modifying not only the position and orientation, but also the linear and angular momentum, because those momentums are reliant on the differences between the current position and orientation p, q and the last frame's position and orientation p^*, q^*, which are kept constant.

9.4 Angle Constraint

Thus, the two bones are pinned together at the point defined in the joint. But with this constraint alone, the ragdoll, which is meant to simulate a human body, will appear impossibly flexible, even contorting its limbs backwards inside its body (as we have disabled collision detection between adjacent bones). An angle constraint can be implemented to keep the relative orientations of jointed bones within reasonable, user-definable limits. The angle constraint defines a default relative orientation between the child and parent bodies plus three values limiting the amount the child can pitch, yaw, and roll away from the default orientation.

TO BE COMPLETED ONCE I FIX THE BUG

10. Stage 1 Prototype Performance

With rigid-body physics and ragdolls implemented we can finally run some tests to benchmark the performance of the Stage 1 game engine prototype's architecture before making modifications. To accomplish this, two demonstration simulations were designed, executed, and performance results were recorded. Each demo was designed to measure how performance of the game engine scales with a varying number of objects.

10.1. Bouncing Balls Demo:

This first demo assesses the performance of the game engine in running the rigid-body's update function as well as the collision detection and resolution algorithms and the function to render each object's SceneNode in OpenGL to the Pygame window. In this demo, X instances of the RigidBody class are created with random initial velocities, sizes, and masses, a coefficient of restitution of 0.95, and they are sphere shaped. The balls begin suspended in the air in a spiral formation, and move along their set initial trajectories as the simulation iterates. They fall due to the force of gravity into an open-lidded box and collide with the walls and floor of this box as well as each other, eventually losing energy and settling, and then the demo is closed. To benchmark the performance of the game engine in running this demo, the maximum and minimum FPS were recorded using Pygame's Clock class' 'get_fps' function, which calculates FPS using the average frame time over the most recent ten frames.

TABLE 1: Performance Benchmark of the 'bouncing balls demo' on the Stage 1 prototype.

Test No.	No. of Rigid Bodies	Max FPS	Min FPS
1	0	59.17	57.80
2	10	59.17	56.17
3	20	60.60	47.84
4	30	59.52	35.58
5	40	46.94	24.27
6	50	36.36	19.68
7	60	31.05	15.12

Table 1 above displays the benchmarked results of this demo run on the Stage 1 prototype game engine. As the number of balls initialized in the scene increases, we see a dramatic decrease in performance, with maximum FPS dropping by fifty percent and minimum FPS dropping by seventy-five percent with only sixty rigid bodies. One other observation worth noting that cannot be shown in the above table is that that maximum FPS was often recorded at the beginning of the simulation and the minimum FPS was often recorded at the end of the simulation. This is because as the balls lose energy and settle on the floor of the box, the number of collisions per frame increases because each ball makes contact with the floor, meaning they collide each frame.

10.2. Stacked Ragdolls Demo:

This second demo uses the ragdoll objects, which are each made up of twelve connected rigid bodies. It therefore assesses the performance of the game engine architecture in performing all the same tasks as were involved in the previous Bouncing Balls Demo, but additionally, also assesses the performance of the constraint solvers in the ragdoll's RigidJoint class. This demo presents the same open-lidded box as seen in the Bouncing Balls Demo and has X instances of the Ragdoll class stacked in the center floating above one another (i.e., not close enough to touch and cause collisions). Over time, the ragdolls fall due to the force of gravity and collapse onto the floor of the box and on top of one another, forming a pile of connected rigid-body objects all colliding with each other and the box. The demo was closed once all ragdolls settled. Like with the previous test, we recorded both maximum FPS and minimum FPS observed through Pygame's Clock class' 'get_fps' function to benchmark the engine's performance.

TABLE 2: Performance Benchmark of the 'ragdolls demo' on the Stage 1 prototype.

Test No.	No. of Ragdolls	No. of Rigid Bodies	Max FPS	Min FPS
1	0	0	59.17	58.47
2	1	12	59.52	37.17
3	2	24	55.86	16.28
4	3	36	40.16	7.86

5	4	48	30.48	5.38
6	5	60	24.69	4.05
7	6	72	23.86	2.84
8	7	84	21.23	2.71
9	8	96	18.93	2.75

Table 2 above shows the benchmark results recorded when the ragdoll demo was run on the Stage 1 prototype. In these simulations, note that the maximum FPS was often recorded in the first few instances of the simulation before the first ragdoll has been able to collide with the floor, and the minimum FPS was often observed at the tail end of the demo when all the ragdolls are collapsed on the floor and piled together. This is because the number of collisions that need to be resolved when the rigid bodies are suspended and not colliding is zero and increases to its maximum as the bodies hit the floor and pile together.

Comparing these results to the results of the Bouncing Balls Demo, we can measure a distinct difference in performance benchmarks from simulations with a comparable number of RigidBody objects. For example, test number seven of the Bouncing Balls Demo (see Table 1) and test number six of the Ragdoll Demo (see Table 2) both involve sixty rigid bodies, but the maximum and minimum FPS recorded in the former test are considerably higher than in the later one (the difference in maximum FPS is 6.36, the difference in minimum FPS is 11.07). There are two reasons for this discrepancy. Firstly, the second demo utilizes ragdolls and so it must run the ‘constraintUpdate’ function of the RigidJoint class in addition to all the other work that goes into the frame update that is also present in the first demo. Secondly, the first demo better disperses the rigid bodies around the box, so they rarely collide with other rigid bodies, especially at the tail end of the simulation when they settle. By contrast, the second demo ends the simulation with most of the rigid bodies piled in a clump at the center of the box, so the observed discrepancy, especially with regard to the minimum FPS recorded at the end of the simulation can be accounted for by these excess rigid-body-by-rigid-body collisions.

An additional observation made in the Ragdoll Demo was that some ragdolls were able to have some of their limbs pass through the floor of the box due to a well-known collision detection bug called *tunnelling*. In a discrete collision detection algorithm, collisions that happen ‘between frames’ are not observed. Tunnelling therefore may occur when one object passes through an obstacle in its path because it moves through and past the obstacle within a single frame, due to high velocity and/or an unusually long timestep. It is observed here because as the number of ragdolls is increased, the number of collisions increase in the later moments of the demo, which causes the time it takes to update the game to increase considerably, and so, some of the rigid bodies of the ragdolls that are still falling in the later instances of the demo can tunnel through the floor of the box within a single frame and avoid collisions.

…

11. Property-Centric Architecture

12. Compute Shaders

13. Stage 2 Prototype

14. Findings

15. Conclusion

References

[Baltman and Radeztsky, 2004]
https://ubm-twvideo01.s3.amazonaws.com/o1/vault/gdc04/slides/using_verlet_integration.pdf

[Bourg and Bywalec, 2013]
see book

[Lovrovic, 2015]
https://www.gamedev.net/tutorials/programming/math-and-physics/capsule-inertia-tensor-r3856/

Appendix