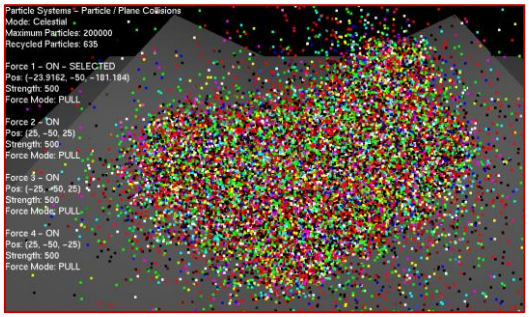
REAL TIME PHYSICS – CUMULATIVE SUBMISSION

A video showcasing the individual labs can be viewed here: http://youtu.be/plcgaf3gHkA

LAB 1 – PARTICLE SYSTEM



The first lab focused on creating a particle system that could interact with a plane. My system supports thousands of particles, even being capable of handling 200,000 particles at once while retaining a smooth 60FPS update, or over 400,000 particles at 30FPS.

The demo consists of three modes, two of which are inspired by the Karl Sims paper discussed in the lectures. The first mode implements forces as single points in the world which can either pull or push particles with a given strength, which causes them to orbit around in unique patterns. The forces are positioned above three oriented planes, which the particles also collide and rebound against. The forces can all be manipulated independently. The user can change their position, their strength, and if they attract or repel the particles. The forces can also be disabled, one by one, to leave only gravity.

The particles are represented as discussed in the lecture, with each having their own position, velocity, mass and age. These values are stored between frames, and each update the accumulated forces are calculated and used to determine the particles subsequent velocity. The particle then moves according to its velocity, multiplied by a time delta to keep the visuals smooth.

Each particle is checked to see if it is colliding with any of the planes in the scene using the plane collision methods discussed in the lecture, and if necessary a collision response is applied to rebound the particle. An impulse is calculated for a colliding particle which takes its velocity in the normal direction of the plane and scales it by a coefficient of restitution to give the correct rebound, which will have lost some energy due to the collision. The velocity in the tangential direction is also scaled by a frictional value, so some energy is also lost as the particle drags along the surface of the plane.

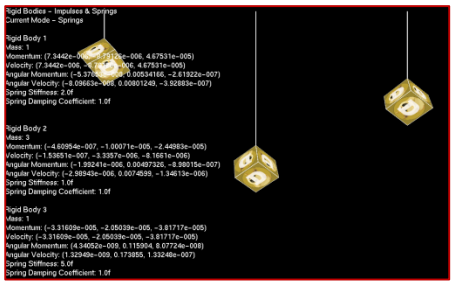
Once collisions have been resolved then the velocity is slightly dampened, to represent air resistance, and then the impulse forces from each of the additional forces in the scene is calculated. The strength of a force is determined using a fall of, which is one over the squared distance between the particle and force, and this force will either pull or push a particle depending on its attract/repel setting.

The second mode implements a snow scene, showing how the parameters of the particles can be varied to give different effects. To represent how snow “sticks” to the ground, rather than bouncing, collisions have no rebound energy at all. Additionally, particles now age to represent melting, and are recycled for a continuous snow fall, which also conserves memory usage. A wind parameter controls the direction of drift, but to make each particle unique a small random offset is applied on top of this.

The final demo is similar to the first, but again changes some parameters to produce an entirely different result. In this case, the frictional force is greatly increased and requires a much stronger force to pull the particles along the planes. This then causes them to shoot upwards in a colourful plume when the force becomes strong enough to lift them away from the ground.

To further optimise the demo, the mid-point integration method discussed in the lecture was implemented, as it proves to be more stable and efficient than Euler for larger timer steps.

LAB 2 – RIGID BODY UNCONSTRAINED MOTION



For the second lab I created a generalised representation for my rigid bodies that could be used in all subsequent demos. The rigid body states hold primary variables concerning position, momentum, orientation and angular momentum, as well as secondary variables such as velocity, spin, angular velocity and the world transformation matrix. It also holds the static values concerning mass and the inertial tensor, as well as their inverses.

The body state also has functions concerning the integration of the forces each frame and the updating of the bodies' force and torque. To further improve on the mid-point method used in the first lab, I implemented Runge-Kutta Order Four to further increase the accuracy of the physics simulation. RK4 samples the derivatives four times, compared to just the one sample used in regular Euler integration. This proves to be particularly useful when implementing springs, also shown as part of this rigid body demo, as simulations can explode very easily when integrating spring forces using traditional Euler while they remain considerably more stable with RK4.

Another method for keeping the physics simulation stable is by using a fixed time step, ensuring that it always updates at the same rate regardless of the current rendering frame-rate. For this demo, and all subsequent ones, a fixed physics time step of 120x a second was implemented. This was achieved by using an accumulator that "gains" time each update frame, and when the accumulator is greater than our given time delta (e.g. 1/120) we update our physics In a single frame the elapsed time could be 1/60th of a second, 1/30th, or any fraction in-between, so the physics could update once, twice or potentially even four times in between each render to the screen. This means that fluctuations in the display rate don't potentially destabilise the physics, and it remains smooth at all times.

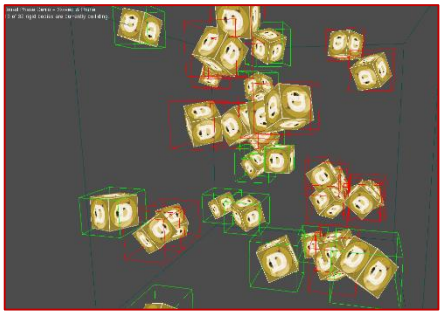
In the first mode of the demo, the user can use the keyboard to apply both traditional and angular momentum to a cube, and watch as RK4 integration updates both velocities accordingly. The second demo uses springs to control the rigid body motion instead. The springs are affixed at a given point and then attached to a corner of the cube, with the transformed vertices being calculated and stored so they can be used to figure out the relative velocities of the cube depending on where the spring has been attached to on it. The linear force of the spring is calculated using Hooke's Law, while the angular force is then calculated by crossing the displacement vector, which is the distance from the centre of mass to the point on the object where the spring is affixed, against the linear force. This creates the scenario, as in the image to the right, where the cube correctly rotates as it gets pulled up by the spring while also being pulled down by gravity.

The final mode only has a single rigid body and a spring, but by clicking the mouse the user is able to move the cube about the scene and release it to see how different positions and velocities impacts the spring and movement of the rigid body.

The lectures notes proved to be a useful source of information for the rigid body state and its implementation, but this was further built upon using Glenn Fiedler's series on Game Physics[1] which discussed the theory behind implementing a useful RK4 integrator as well as simple spring physics.

[1] http://gafferongames.com/game-physics/physics-in-3d/

LAB 3 – BROAD PHASE COLLISIONS



The third lab focused on the first stage of collision detection, which involved creating a large scale broad phase check that could be used to prune the amount of checks needed in the more expensive, but also more precise, narrow phase. I implemented the Broad Phase checks using Axis Aligned Bounding Boxes (AABBs), as they proved to be more accurate than using bounding spheres but were still very quick to create. AABBs don't handle rotation well, so need to be updated each frame, but only require six floats (min / max vectors in each direction) so have a low memory footprint. It is also very computationally efficient to check two AABBs together to see if they are overlapping, so they seemed to strike the best balance between complexity and accuracy.

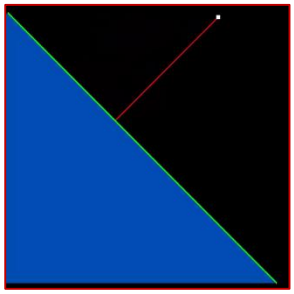
The AABBs were easily implemented using the information from the lecture slides, and could be updated each frame based on the transformed vertices of each rigid body, which was created as part of the second lab. These AABBs could then be drawn on the screen, and coloured to indicate if a collision was occurring or not.

However, although AABBs are computationally efficient it would still be an expensive O (N^2) operation to check each AABB against every other AABB in a scene to see if they overlap. To solve this, Sweep and Prune was implemented to perform these checks instead. Sweep and Prune splits the scene into three, one for each axes, and checks the min/max values of each AABB in that axes to see if they overlap. If two AABBs overlap on all three axes, we consider that a broad collision and can proceed to narrow phase for more detailed checks.

To ensure Sweep and Prune was as efficient as possible, an Insertion Sort is used to sort the axes each frame. The axes are stored between frames, so the insertion store exploits frame coherency to only make small changes to the rigid bodies that have moved. A table of collision pairs, e.g. (1, 2), (1, 3), (1, 4) and so on, is kept to track the number of overlapping axes for each potential pair. The table is created as a nested array, so has constant access, and can be quickly iterated over to find overlaps. This iteration also resets the array for the next frame, removing the need for a second "reset" iteration at the start of each frame. The iteration follows the sequence of triangular numbers, and for ***n*** rigid bodies in a scene only requires ***n(n-1)/2*** checks, not ***n^2***, as we exploit the fact that if A is colliding with B, then B is also colliding with A and don't perform the same check twice. Additionally, if no collisions are found on an axes then we know that no objects are overlapping, so we can quit out early to save on needless computation.

Although it could be further optimised, e.g. frame coherency could be better exploited so the insertion sort itself actually updates the collision pairs table if overlaps change[2], the current Sweep and Prune approach proves to be blisteringly fast and can handle hundreds of objects at once, with rendering the objects becoming a bottleneck before the Sweep and Prune sort does.

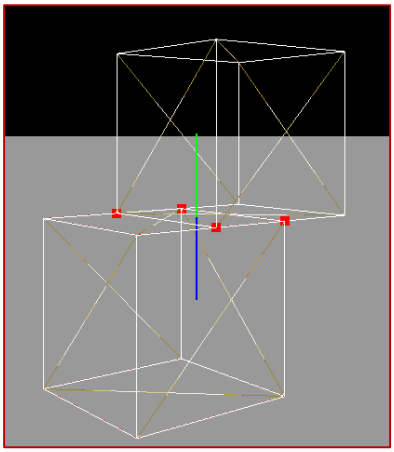
LAB 4.1 – VORONOI REGIONS



As an aside to the core narrow phase collision for lab 4, I implemented a simple demo that showed how Voronoi Regions could be used to find the point on a triangle that was closest to a given target point. This test would also identify if the closest point was along an Edge, on the Face, or was one of the Vertices making up the triangle. This demo was implemented entirely using the notes provided within the lecture slides.

[2] http://www.codercorner.com/SAP.pdf

LAB 4.2 – NARROW PHASE COLLISION



Building upon the broad phase from the previous lab, the next step was to create a comprehensive narrow phase that could definitively detect if two rigid bodies were truly colliding. I initially explored using the Glibert Johnson Keerthi (GJK) algorithm, but while researching how to implement that I came across another article on using the Separating Axis Theorem (SAT) to detect collisions in 2D[3]. The theory was easy to understand, and discussions in the comments suggested methods for easily extending it into 3D.

SAT works on the theory that if two convex objects are not colliding, then there is a least one axis in which the projection of the two objects will not be overlapping. In 3D, the axes which must be tested are the unique crossed axes of each edge in a rigid body against every edge in the other rigid body. Naturally this can create a large amount of potential axes, but the reality is that the vast majority of them will be either parallel or otherwise duplicate. Optimisations in the creation of the axes can prune the duplicates, and we can even optimise the axes checks to only check unique edges. For example, a cube has 12 edges, so two cubes would create 144 different axes of potential separation. Although only three edges on each cube are actually unique so this can be reduced to 15 separation axes for two rotated cubes, or only 3 axes for two cubes which share the same rotation.

Once the axes have been uncovered, each rigid body is then projected on to the axis and their minimum and maximum points are recorded. A simple check of min values against max values can then tell us if objects are overlapping, and by how much. An additional benefit to this search is that if the two objects are overlapping then the axes with the minimum overlap is actually the collision normal, so checking for overlap naturally returns the collision normal too.

Having gained our collision normal, we can then determine which face / edge / vertex on each object is actually involved in the collision and create a collision manifold. This is done through a process known as Clipping, and again a very good resource that clearly explained the concept in 2D was used as a base[4], although the Sutherland-Hodgman algorithm was used to implement this in 3D[5]. You find the face on each objects that is involved in the collision and create an incident face from one rigid body and a set of reference faces from the other. You then clip the vertices of the incident face against each of the normals of the reference faces and the final result is a contact manifold with all the collision points for the current collision.

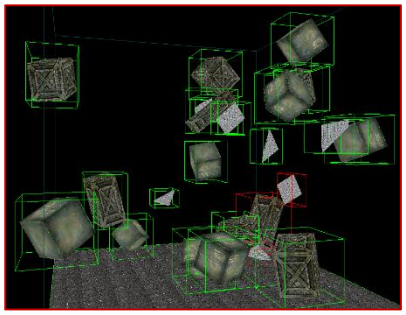
Similarly to the SAT process, there is a potential for a lot of duplication in clipping, especially when dealing with complex objects with many faces and triangles. To solve this, I created a simplified “collision mesh” representation of each rigid body type, which is separate from the more detailed meshes used for rendering. This collision model has some simple information about the vertices and faces that are to be used in a collision, and this greatly simplifies any calculations. The combination of SAT and Clipping handles any generic rigid bodies, and allows for object-object collisions as well as plane-plane collisions. While it might not be as efficient as GJK, the optimisations to reduce the amount of duplicated axes and collision data means it has no problem handling multiple objects colliding at once, especially when the narrow phase is only executed after a broad phase collision has occurred.

---

[3] http://www.codezealot.org/archives/55
[4] http://www.codezealot.org/archives/394
[5] http://en.wikipedia.org/wiki/Sutherland–Hodgman_algorithm

LAB 5 – COLLISION RESPONSE

The final lab combined everything from the second, third and fourth labs to create a physics engine where the rigid bodies correctly react to each other. The main implementation for this lab was the impulse response, and the equation used for this was the Impulse Magnitude one discussed in the lectures. This was combined with the readings from Chris Hecker, as well as the notes in the Multiple Contact Resolution lecture with regards to penalty methods, stability issues, simultaneous contacts and the iterative LCP solver. In addition, presentations made by Erin Catto at GDC discussed similar methods for using iterative solvers to handle collision response using impulses[6].

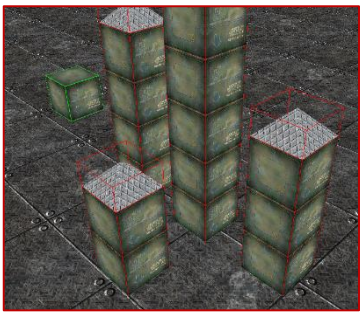


Essentially, iterative impulses builds upon the information gained from our previous Narrow Phase lab. We have our collision normal, which is the direction we want to push our colliding objects along to separate them, and we have a series of points which form the contact manifold and tells us how the objects have collided. We can apply impulses to these points to push the object, and depending on where these points are on each object this will create both linear and angular movement, making the objects bounce and spin as you would expect.

Applying the forces incorrectly can cause the simulation to quickly explode, so the solution is to iterate over each point in the collision manifold and apply an impulse relative to the point's penetration depth and current velocities. Applying a force to one point will cause a change in the velocities of the other points, and so the iterative approach is needed to ensure that the forces acting on all points converge to push the object in the right direction.

Initially this was done on a per-collision basis, as narrow phase detected collisions, but this quickly caused the simulation to explode when multi-object collisions occurred. This problem was solved by storing all the collision pairs for each frame, and then solving for all collisions at once.

Depending on the complexity of the collision, the impulses can converge quickly or slowly, and so this approach trades accuracy for speed of execution. As an optimisation, if the impulses on a collision have managed to converge early then we quit early and stop working on that pair. With a low number of iterations, errors can build up over time. This usually isn't a problem for two objects colliding against each other, but the errors can be more easily noticed if you observe stacked objects as they will eventually slide apart and collapse. In general, however, optimisations in the previous stages meant more time could be spent on this stage, allowing for a higher iteration count that minimises the build-up of errors.



The collision response is further optimised by only recalculating the information it needs, so the values in the impulse magnitude equation which are fixed for a given frame are only calculated once, while those that are changed with each iteration are calculated multiple times.

When it comes to objects resting, either against each other or on planes, the impulse magnitude equation was expanded to provide both a restitution impulse and a penetration penalty impulse. This stops objects "sinking" in to each other, and by allowing a small amount of penetration, e.g. a "slop", the jitter effect that can sometimes occur is also prevented. The combination of all these features creates a robust physics engine which can handle quite a lot of scenarios, while still retaining a smooth framerate with good overall performance.

[6] https://code.google.com/p/box2d/downloads/list