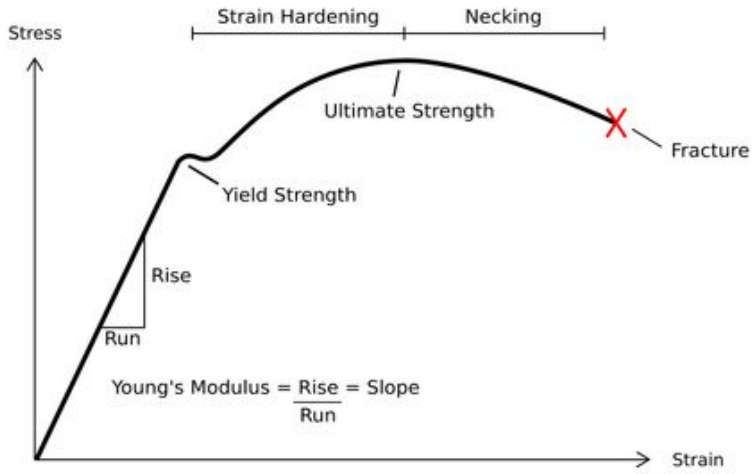
Plastic and Elastic Deformation with Spherical RigidBody

Initial Research:
My initial research started by reading “Elastic and Plastic Deformations with Rigid Body Dynamics” Jeff Budsberg et al[1]. This was a great starting point as it clearly broke down the main elements to the technique. Firstly decomposition, converting the input mesh into a network of spherical rigidbodies, this is done by using a sphere packing algorithm and using the resulting sphere data to generate rigidbodies. Modelling material properties is controlled via the use of constraints.

Plastic and Elastic Deformation:
Elastic deformation is a change in shape of a material at low stress that is recoverable after the stress is removed. This type of deformation involves stretching of the bonds, but the atoms do not slip past each other. When the stress is sufficient to permanently deform the shape of a material, it is called plastic deformation. Plastic deformation involves the breaking of a limited number of atomic bonds by the movement of dislocations[2]. Deformation of a material is linked to Stress and Strain, every material has a Stress/Strain curve which can be used to determine characteristics of the type of deformation a material may undergo at specific stresses. This curve can also categorize materials into 2 distinct types, Ductile and Brittle.



Stress (Pa) = **F/A** where **F** is the force (N) being applied. **A** is the cross-sectional area of the shape (m^2).
Strain = **delta L/L** where **delta L** is the change in length (m) and **L** is the original length (m).
Linear elastic deformation can be governed by Hooke’s Law: **F = -delta x * k** where **delta x** is the change in length and **k** is the spring constant.
Elastic deformation stops when the Stress/Strain curve reaches the material’s Yield Strength, from this point onwards plastic deformation is occurring[4]. In code we can recreate this effect using Bullet Constraints, when the Yield Strength is reached we can dynamically change the constraint properties to represent the new material properties thus emulating

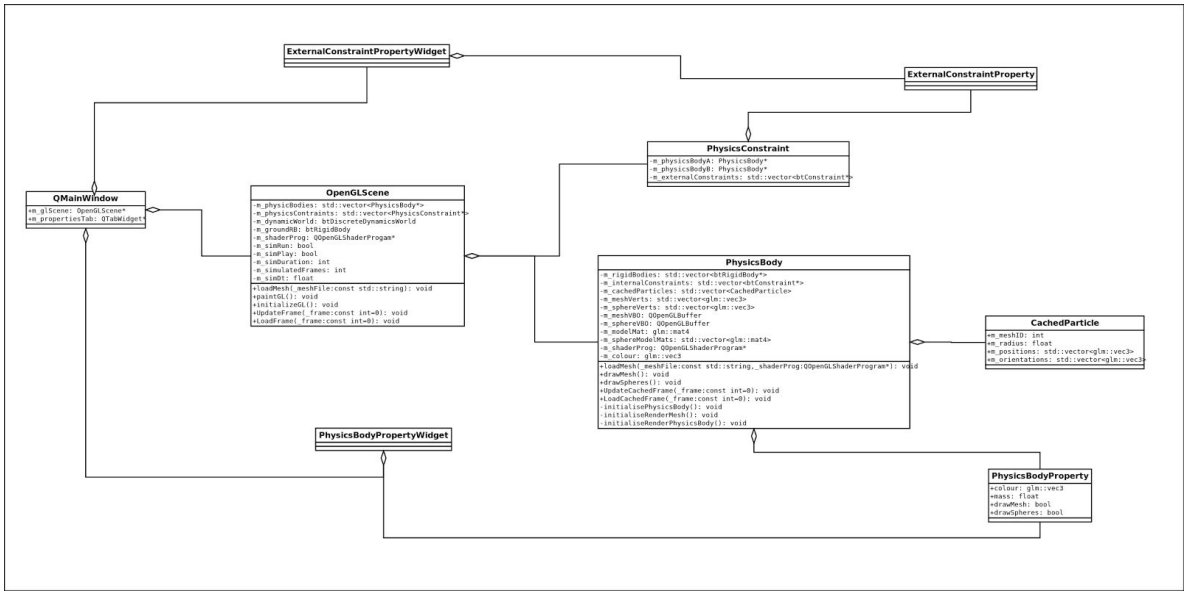
plastic deformation. After some threshold strain we can break the constraint representing fracturing.

Rigidbodies and constraints:
Bullet physics library is a well established realtime physics library, it provides rigidbody physics and offers various constraints, as a result I decided to use it.

Sphere packing:
The OpenVDB library was mentioned as a useful library that could be used for this project so taking a look into it I found it offered its own sphere packing implementation. It requires the mesh to be in a volume format before packing with spheres. I decided to use OpenVDB to complete this step in my initial proof of concept. If I get the rest of the project completed with time to spare I will look further into implementing custom algorithms and testing comparisons.

UML Design:

The following UML diagram shows my initial design for how I plan to implement this system.



The main classes are:

OpenGLScene inherits QOpenGLWidget to enable easy openGL rendering. It holds a pointer to the btDiscreteDynamicsWorld instance and has a vector of PhysicsBody's and ExternalConstraints.

PhysicsBody is a class that handles the rendering and simulating of single mesh.

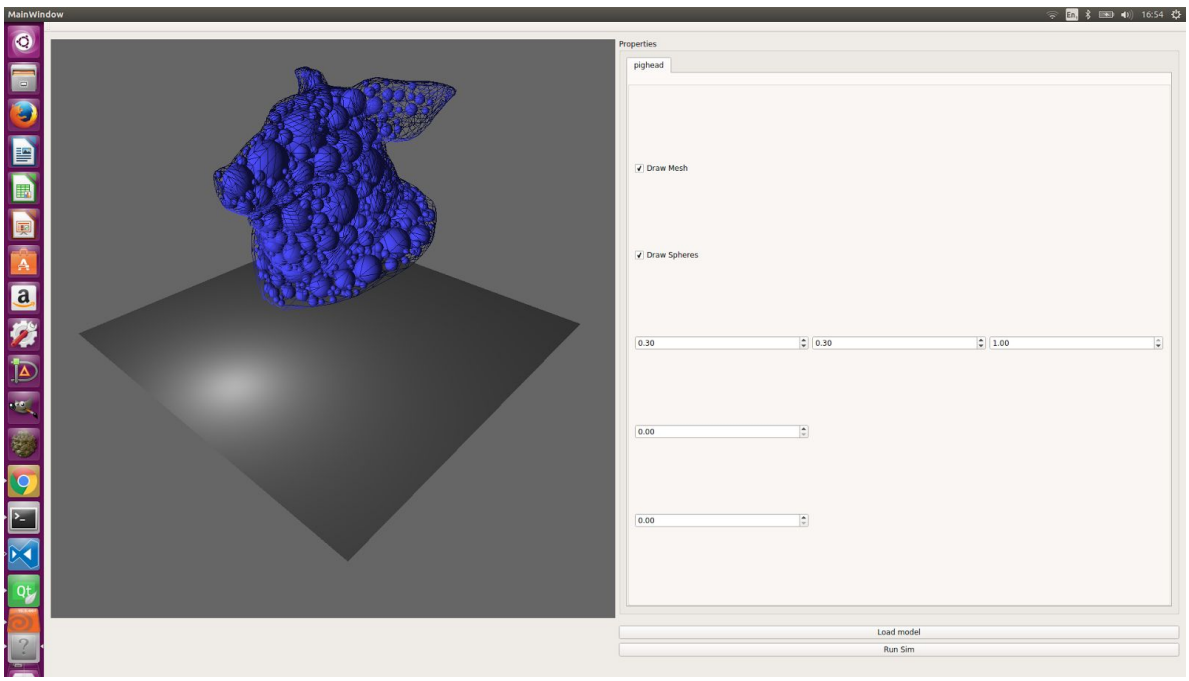
PhysicsBodyPropertyWidget enables tweaking of physics properties via a UI.

ExternalConstraint is a class that handles the constraints (glue) between meshes.

ExternalConstraintPropertyWidget allows tweaking of constraint properties via a UI.

Framework:

I have set up the application in Qt. This is the current state of the application:

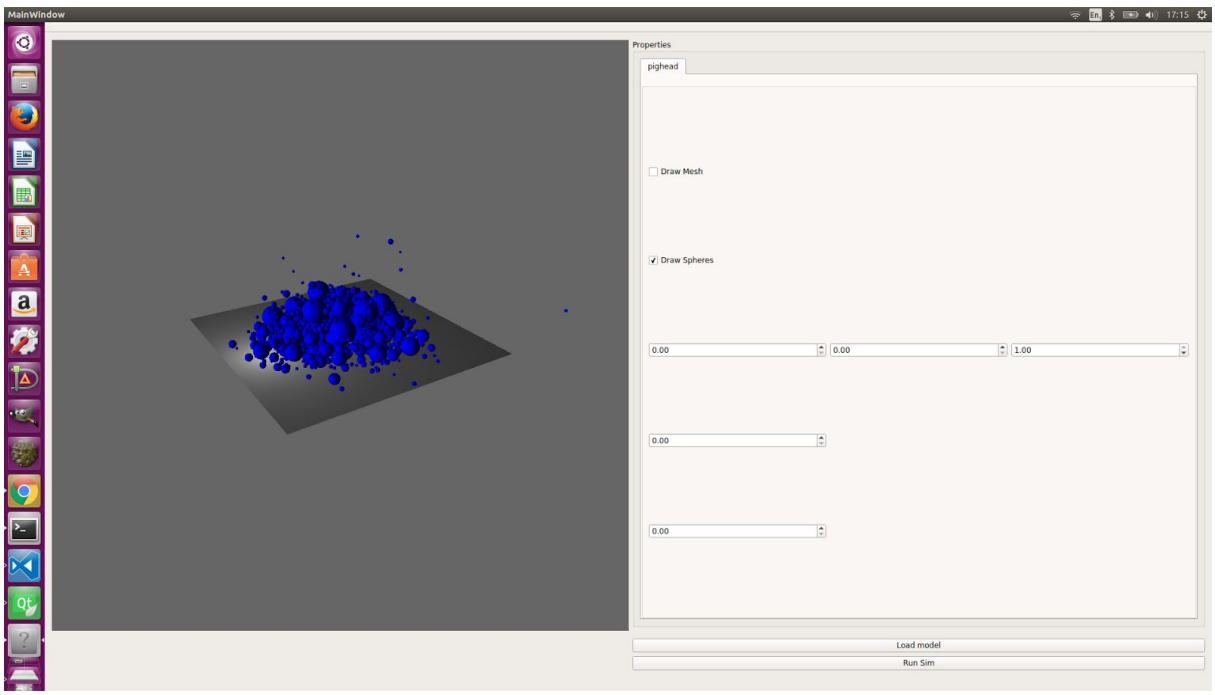


Loading meshes:

I then started looking into loading in common graphics file formats. I decided to try and get Alembic to load in as this is a common format for caching simulations. I successfully got a basic Alembic Teapot model loading and rendering to screen however it seemed to be very particular about which package generated the Alembic file. Due to this inconsistency I decided to also support loading of other common 3D formats such as .OBJ using the ASSIMP library.

Rendering meshes and spherical representation:
Then I started designing the system a bit more, and implemented a PhysicsBody class which would handle the loading, rendering and simulating of individual meshes.
In the PhysicsBody class I load in a mesh. I then use openVDB to generate a volume from the mesh and fill the volume with spheres. I extract the sphere information and generate sphere meshes for rendering to check everything is working correctly.
As the number of spheres that fill up the volume will be dependant on the user, so could be large for high detail, I decided to use instance rendering[4] for all the spheres in a single simulated mesh. This vastly reduces the total draw calls overall, with each mesh only being one draw call and the poly count being that of 1 sphere per simulated mesh, thus improving memory efficiency on the GPU.

Integrating Bullet Physics:
Once I had the rendering pipeline roughly sorted I started integrating Bullet Physics. I made each sphere generated into a btRigidBody, and stored that rigidbody in the m_rigidBodies attribute of the PhysicsBody class, this way each spherical rigidbody associated with a particular mesh is kept together and I can easily add them to the dynamic world with correct collision masks. I then started naively adding bullet constraints between the internal spherical rigidbodies of a mesh, this yielded some crazy results as I had not tuned the constraints appropriately. I am currently looking into Bullet constraints in more detail now to create the correct behaviours.



TODO:

Fine tune internal constraints so they default to a useful state.
Add external constraints between PhysicsBodies (separate meshes).
Develop/improve UI to allow for easier tweaking of simulation parameters, e.g. per PhysicsBody, per external constraint, physics world params.
Sphere to vertex weighting for re-skinning after sim.

Stretch goals:
Implement a caching system, to allow for faster playback after simulation has been run, and for exporting to DCC's.
Implement keyframing of physics parameters for high fidelity of simulation.

All up to date code and UML diagrams can be found in my Github repo:
https://github.com/IdrisMiles/masterclass