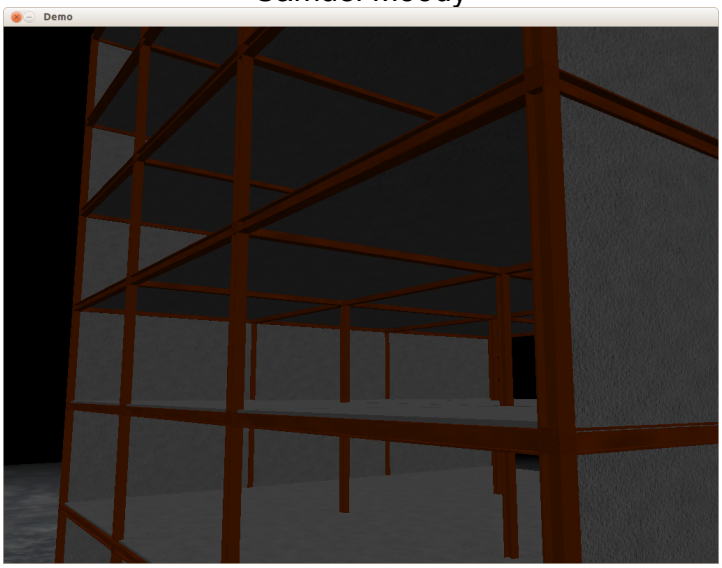
Interactive Structural Demolition

Kevin Law
Samuel Moody



Abstract

We present a method for a structural simulation of destructible 3-dimensional metal frame buildings at interactive frame rates. Beams, floor slabs, and wall sections are simulated as rigid bodies and joined together by rigid constraints. Each constraint is given a set of breaking thresholds for the force and torque around each axis, allowing them to fail if the structure is overloaded or damaged. Additionally, we investigate several ways to fracture a thin surface.

Keywords: Demolition, Destruction, Fracture, Structural Analysis

1 Introduction

Many 3D video games offer some form of environmental destruction. However, typically it is limited to certain objects, such as explosive barrels, breakable crates, and to specially-made wall sections or obstacles. The majority of the level geometry is indestructible. In games where larger areas can be destroyed, the destruction feels scripted. For example, the buildings in Battlefield: Bad Company 2 are very predictable. Most buildings can be collapsed by destroying enough outer wall sections. These wall sections are discrete all-or-nothing parts which are not physically simulated. The interior walls are indestructible until the whole building collapses.

We classify destruction into two types: structural and detailed. Detailed destruction concerns small-scale objects such as those mentioned above. Structural destruction includes major features such as walls, bridges, buildings, and terrain. Detailed destruction is commonplace; the explosive barrel is ubiquitous in modern first- and third-person shooters. We thus focus primarily on structural destruction.

Our goal is to create a destructible building that responds physically to the user's actions, rather than by applying a set of predetermined destruction patterns. To this end, we model a building as a semi-rigid frame and add walls and floors.

To provide additional realism and visual detail, the walls and floor should fracture when subjected to sufficient stress. Although we did not have time to combine the frame simulation with a fracture system, a few fracture algorithms were investigated.

1.1 Related Work

There is plenty of research that has been conducted on the analysis of structures. Such knowledge is integral to the design of modern bridges, buildings, and other structures. In particular, static analysis is widely used to determine the load on individual members of frames. [1] describes a method for applying static analysis to statically indeterminate frames, which cannot be solved using regular methods such as the Method of Joints.

One of the most famous building collapses is that of One and Two World Trade Center. On December 1, 2005, the National Institute of Standards and Technology published a report detailing the causes of

the collapse of the Twin Towers and an analysis of the Twin Towers' internal structure. The information in this report was used to create a realistically accurate simulation of building demolition [2].

2 Structural Simulation

2.1 Overview

Our initial plan was to create a static structural analysis system for the static parts of buildings and use a rigid body simulator for debris. However, as the structure is damaged, entire sections may move while still intact. These sections still need to be simulated properly, including responding to collisions with other parts of the same original structure. Additionally, the structures being simulated are not statically determinate, which means simple statics algorithms are not applicable. The structures are usually over-constrained initially, and may become under-constrained as they are damaged [3].

A fully dynamic system can solve all three cases if some deformation is permissible. In fact, since real structures deform under load (for instance skyscrapers swaying in the wind), it is potentially more accurate. Since implementing a complete rigid body dynamics system (including collision detection and constraints) is beyond the scope of this project, we used an open-source software package for this task. The Bullet Physics library was chosen because it is intended for games (and would thus be more likely to yield good performance) and because we were familiar with it.

2.2 Construction

We create rigid bodies for each structural element of the building and join them together with rigid constraints. First, we create posts (vertical beams) in columns and constrain each post to the post below it. Next, horizontal beams are attached between post joints and are constrained to the lower supporting post. As shown in Figure 1, floor slabs are attached to groups of 4 horizontal beams forming a square, and wall slabs are attached to upper and lower horizontal beams and left and right posts around the outside of the building.

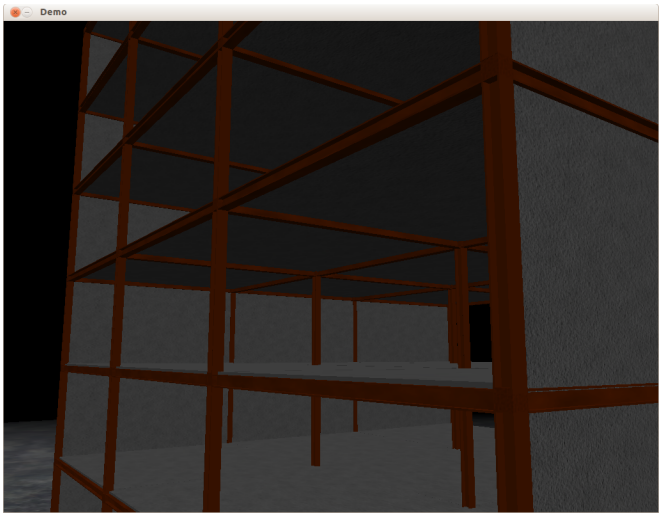


Figure 1: Layout of beams, posts, and floor and wall slabs.

Each joint is given a breaking strength in both force and torque. We specify these values as 3-vectors multiplied by an overall strength constant to make the directional components clear. Posts are given high vertical strength but poor horizontal strength, while beams are given more horizontal strength because they need to hold the building together.

2.3 Projectile and Vertical Post Behavior

Research on building collapses show that vertical post failure occurs when the horizontal supporting beams anchoring them to other vertical posts, thus providing stability, are removed by some damage sustained by the building. Once the supporting beams are removed, the posts begin to bulge horizontally as they weight they support bears down on them. Eventually, the posts snap, causing the portion of the building that they were supporting to collapse.

In the simulation, the projectile performing the destruction is a small ball. The joints connecting the vertical posts and their associated joints connecting the horizontal beams to the vertical post joints are stored in a hash table. When a horizontal beam joint is disabled by either too much stress on the joint or impact with the projectile, an event handler in the program will check the associated vertical post joint to see if other associated horizontal beam joints are still present. If they are not present, the program will disable the vertical post joint, mimicking the real-world behavior of a vertical post snapping due to a lack of horizontal support beams. This behavior in the Twin Towers can be seen in Figures 2 and 3 [4].

Figure 2: Failure of horizontal supports in the Twin Towers due to fire weakening the joints.

Figure 3: Snapping of vertical posts in the Twin Towers without horizontal supports.

2.4 Debug Visualization

To aid debugging and tuning, a visualization system was created. Since the graphics engine being used did not easily support changing material colors per-instance, lights were used instead. The lights are positioned near the joint they are indicating, and offset in front of the beams so that they illuminate them as shown in Figure 4. They may be enabled for posts, beams, or both, and to aid visibility, a different model and no texture are used on members where lights are enabled.

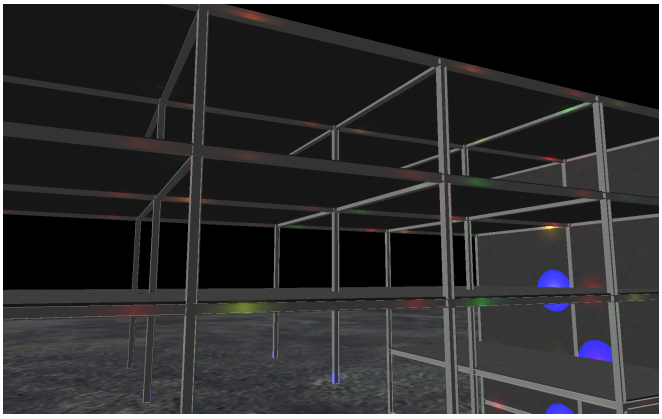


Figure 4: Visualization of forces, torques, and areas of joint failure.

The lights' brightness increases with joint stress. The green channel indicates torque while the red channel indicates force. The lights can either be removed when a joint fails, or the blue channel can be activated so that broken joints are easily identifiable.

3 Fracture

3.1 Overview

While large-scale destruction is the focus of this project, it would not be realistic without the addition of some detailed destruction. In our simulation, the individual beams, floor slabs, and wall slabs were indestructible. In a real building, the walls and floors would crack and crumble, while the beams would bend and twist. While we did not have time to complete a fracture system, a few different 2-dimensional algorithms were investigated.

3.2 Simulation Base

A simple rectangular wall was constructed at ground level with a rigid body applied for collision detection. Each algorithm was tested by running some initialization code on the wall, and then calling a “Break” function when the user hit the wall with a fast-moving spherical rigid body. The contact location was passed from the physics engine to the breaking function.

3.2 Voronoi Fracture

Voronoi fracture is relatively well known and produces good results for isotropic materials like stone, concrete, and glass. Because part of our goal was to create unique effects based on the user's actions, we first attempted to apply a Voronoi pattern dynamically upon impact.

3.2.1 Grid Structure

The initialization code creates a regular 2D grid over the wall surface and assigns each cell a local Voronoi position. The grid provides several advantages:

- It ensures a relatively uniform distribution of points across the wall
- It regulates the sizes of the chunks produced
- It bounds the number of points that can possibly contribute to a given chunk
- It keeps track of where the wall has been destroyed via boolean “intact” flag

When the wall is impacted, the Break function sets the intact flag for the corresponding grid cell to false and calls BuildGeometry, which then constructs geometry

for the entire wall.

3.2.2 Geometry Creation

Due to time constraints the BuildGeometry function was never completed, although most of the supporting code is present. The BuildGeometry function iterates over all intact grid cells within the wall's dimensions. If completed, it would compute all of the edges around the corresponding Voronoi chunk. Only the edges shared with a non-intact grid cell would be added to the geometry since the others would not be visible.

The Voronoi edges for a given grid cell are determined by the following algorithm:

For all grid cells (i,j) from (x-2,y-2) to (x+2,y+2)
    Compute line vector using the Voronoi position of the cell and it's neighbor (i,j)
    Turn the line perpendicular around its midpoint using the cross product
    Clip the perpendicular line against the wall borders
Clip all perpendicular lines with each other to form a convex polygon

Notice that only neighbors within 2 grid cells are queried. That is because it is impossible for a point outside this range to not be clipped out by one of the inner points as shown in Figure 5. Lines are stored as (start, end, normal) 3-vectors. Using 3-vectors despite working in 2D allows for certain conveniences such as using the 3D cross product. The clip algorithm works by finding an intersection between two lines. Either the start or the end of each line (based on the normal directions) is set to the intersection point.

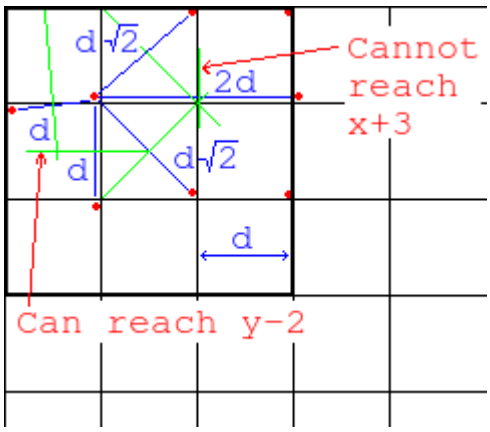


Figure 5: Query of neighbors in grid cells.

The last step is not fully functional at this time, and only clips some of the lines.

3.3 Quadrilaterals

Another algorithm involves connecting grid points to their cardinal neighbors as shown in Figure 6. It is much easier to implement than Voronoi fracture since edges are already defined by the grid cells. The BuildGeometry algorithm simply checks if the edge would be visible (by checking neighboring grid cells' intact field) and creates the necessary geometry.

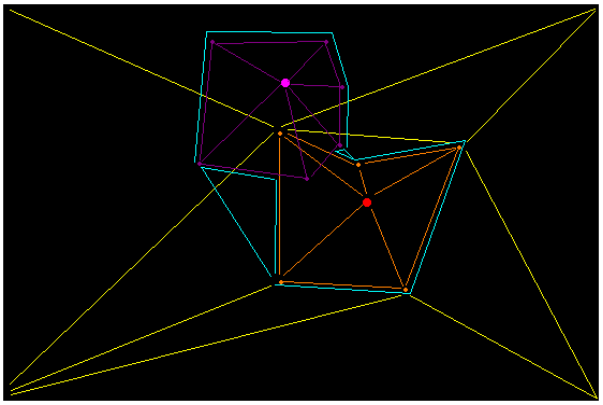


Figure 6: Connection of grid points to cardinal neighbors.

Unfortunately, it suffers from the drawback that grid cells have no thickness. Consider the case where both the left and right neighbors of a cell are destroyed. When generating the mesh, the center grid cell will have zero thickness and must be ignored. Alternatively, the quadrilaterals could be destroyed instead of grid cells. This would require them to store a flag, and it would make determining impact area more complex than a simple rounding operation.

3.4 Local Damage

A third algorithm that does not rely on any precomputed data or patterns was designed. It was inspired by the spiderweb impact patterns formed by concrete and glass. A random number of points are created in sequence around the impact location. The algorithm creates them sequentially by iterating around the impact point in an approximately circular fashion:

```
choose integer n between 3 and maxPoints
repeat i from 1 to n (inclusive)
        set angle = [ i / n + random(-1/2n, 1/2n) ] * 360
        set length = random(minLenth, maxLength)
        addPoint( impactPos + length *
Vec2(cos(angle), sin(angle)) )
```

The points are joined consecutively into edges, forming a polygon. If length is kept within a certain range (dependent upon n), it will be convex. Concave shapes may be turned convex by adding surface triangles in the concave regions:

For each pair of consecutive edges:
    if cross(a,b).length > 0:
        addTriangle(a.start, b.start, b.end)
        flag edges a and b as inner
        go back one edge to check if edges[a-1] and the new edge are concave

Once a convex polygon is produced, the new wall surface can be created by triangulating with the wall border. In the case of multiple impacts, and thus multiple polygons, the polygons can first be triangulated together into one convex polygon.

Due to time constraints, triangulation was not implemented. However, we did successfully create chunks from the polygons illustrated in Figure 7. A new chunk is created for each impact. The wall is not modified.

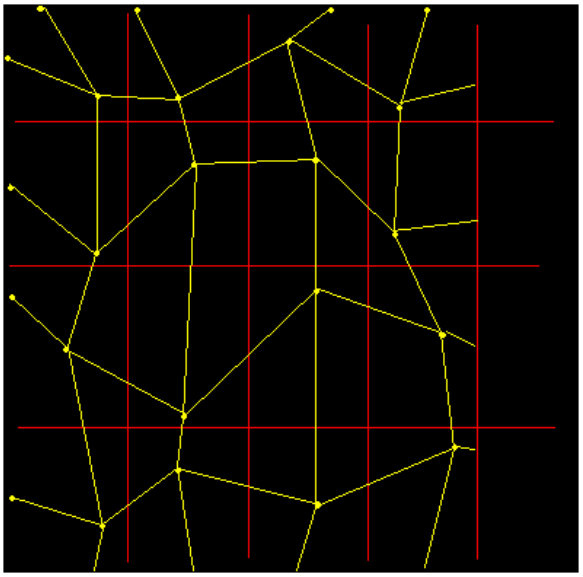


Figure 7: Creation of chunks from polygons.

4 Major Challenges

4.1 Accuracy and constraint weakness

Using a dynamic system led to some problems of its own. Since Bullet is intended for games, it sacrifices some accuracy for performance. Because of this, the constraints are fairly flimsy. This causes some structural problems.

If the breaking thresholds are set too high, the building can bend unrealistically - even to the point where the beams no longer touch each other as shown in Figure 8. Additionally, the building settles a lot from its initial configuration. Besides changing the actual dimensions of the intact building, it induces significant stress on the lower joints. If the joints cannot handle this extra stress, they will break and the building will collapse. Finally, the constraints are also less accurate. Parts of a building that lack support will tend to bend over instead of breaking off. Since the constraints do not apply enough force to keep these parts upright, they also do not exceed their breaking threshold.

Bullet Physics uses a sequential impulse constraint solver by default. To increase the stiffness, we increased the solver iterations from 10 to 200. This yielded much better results – a 3x3 post, 12 story building sagged around 1 meter with 10 iterations, but only around 0.1 meters with 200 iterations.

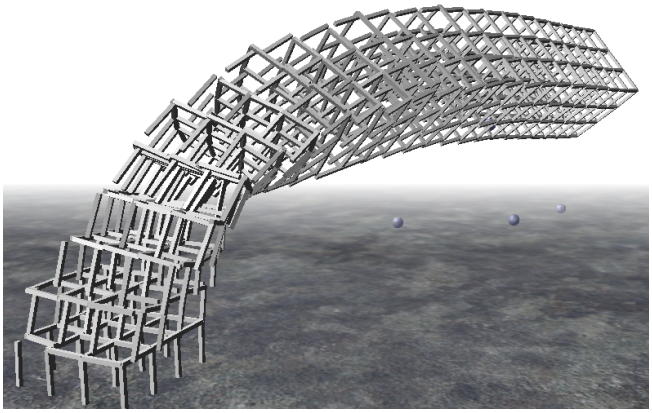


Figure 8: Unrealistic bending in the building.

We also explored alternative constraint solvers available with Bullet Physics, such as the Dantzig MLCP solver. These solvers supposedly make constraints stiffer, but their performance for this task was extremely poor. Attempts to simulate a building of any significant size resulted in excessively long frame times (to the point where it appeared that the application was frozen), so they were abandoned.

4.2 Joint Parameters

The considerable amount of parameters on each joint presents difficulties when trying to determine their optimal values. Each joint has six parameters acting on it: three forces and three torques with each force and torque acting in the x, y, and z directions. Each of these parameters must be adjusted so that the joints and the structure as a whole displays behavior that matches that of reality. However, the large amount of parameters present make it difficult to view the effects of each one individually during the simulation.

To more easily visualize the effects of each force, the program will highlight the ends of beams and posts adjacent to each joint to display the relative magnitude of the force or torque acting on it. The brighter the highlight, the stronger the force or torque is acting on

the joint. A red highlight indicates a force acting on the joint, a green highlight indicates torque, and a yellow highlight indicates both simultaneously. A blue highlight will signify that a joint has failed.

4.3 Performance

The program was tested on two PCs: a Lenovo ThinkPad T420 with an Intel Core i5 2540M @ 2.6GHz running Ubuntu 14.04 x64 and a custom-built desktop with an Intel Core i5 4670K @ 3.4GHz running Linux Mint 17.1 x64. GPU specs are unnecessary since the program is CPU-bound.

When running the simulation with a building that is 4 posts by 5 posts by 6 floors, the Lenovo computer displayed noticeable performance issues and had to be slowed down to an eighth of the normal speed to run the simulation without any frame rate stuttering. At normal speed, the simulation will run at 7 frames per second and show noticeable stuttering. On the other hand, the desktop was able to run the simulation with the same size building with significantly less stuttering. When the simulation is performing physics calculations on the structure in the few seconds immediately after it is struck by the projectile, the frame rate is 15 frames per second. However, after the calculations are complete, the frame rate climbs to 60 frames per second, something that does not occur on the Lenovo computer.

5 Conclusion

5.1 Viability of the Technique

Although unoptimized, the technique presented will successfully simulation the destruction of a building due to damage sustained from a projectile. However, it requires further refinement before it can be used in a commercial video game. These refinements include using a static/dynamic system, implementing fracture for additional realism, and adding performance enhancements.

5.2 Future Work

5.2.1 Combined Static / Dynamic System

To deal with some of the shortcomings of the dynamics system, a specialized solver could be implemented for the structural simulation. Using a specialized solver would be advantageous because it could be tuned for the specific task to gain both performance and accuracy.

As discussed in section 2.1, one of the issues with using a static system was the difficulty in determining which parts are static and which are dynamic. While we did not investigate this further in the interest of time, it is definitely possible. A potential solution would be to use a connected component graph to segment the intact structural members into separate pieces which would then be analyzed separately. Each segment could be its own rigid body and receive forces from the dynamics engine.

5.2.2 Application to Games

The models used so far are very basic and are not directly usable in a commercial video game unless it is a very basic one involving 2D demolition.

One of the challenges in applying this simulation to a real model is that it deforms dynamically. While it may be possible to use standard animation techniques with this system (using the beams as bones), a better approach would be to use a combined static/dynamic system as mentioned in the previous section (5.2.1). The model would be segmented along with the underlying structure, and would thus never deform except by breaking into parts.

Fracture would play a key role here as most models are not just posts, beams, and slabs. In order to convincingly break a model apart, the extra material around the beams would have to be broken. Physical forces could be used to determine fracture locations and thus a separate algorithm for splitting the model would be unnecessary.

5.2.3 Misc / performance?

The most noticeable problem relating to performance is that the program lags on weaker computers. This is because the program is CPU-bound and unable to use resources from the GPU. By optimizing the code to use additional resources in the GPU, the performance of the simulation can be increased dramatically.

Acknowledgements

We would like to thank the developers of the Urho3D game engine and the Bullet Physics library for providing these tools for us and the developer community to use at no cost.

References

[1] D. Schodek and M. Bechthold, *Structures*.
[2] National Institute of Standards and Technology,

'Final Report on the Collapse of the World Trade Center Towers', 2005.
[3] L. Bucciarelli, *Engineering Mechanics for Structures*. Mineola, N.Y.: Dover, 2009.
[4] *Nova: Why the Towers Fell*. WGBH/PBS, 2002.