Team Seven Project Report: Feynstein

Samantha Ainsley Eva Asplund William Brown
Colleen McKenzie Robert Post

November 3, 2012

Abstract

Feynstein is a language designed for making physical simulation accessible to those who want to experiment with physics but do not have significant experience in computer science. Feynstein makes the process of going from an experiment's conception to its simulation simple and fast, and provides the end user with accurate video renderings, either in a file or on screen.

Contents

1 Introduction

Historically, if users wanted to have the power of a full physical simulation engine, they had to either purchase and learn very complex and expensive 3D simulation software, or write it themselves (usually in C++). Because these options are costly and time-consuming, often students of physics will learn simulation using a software package like Matlab or Mathematica. These tools have the benefit of abstracting much of the complexity of the task, but with the loss of complexity comes a loss of control. Feynstein attempts to be a midpoint between these two extremes; it is a language that allows for the full expressivity of a complex simulation package, but with the ease-of-use that comes with a tool like Matlab.

Feynstein's native simulator and its simple and convenient syntax make it an ideal tool for researchers with limited programming experience for configuration simulations to test experimental hypothesis. Moreover, the language offers an excellent framework for teaching the fundamentals of physics-based computer simulation to students with limited understanding of mechanics or programming.

Feynstein is fully object oriented, allowing for easy extensibility. Users can customize integration and rendering methods, giving them full control of how their animation is calculated and displayed. Finally, users can use the built-in forces and 3D solids and define their own. Feynstein is appropriate for all levels of user expertise, whether they want to simulate a sphere on a spring or a complex compression force on a custom shape.

2 Language Tutorial

2.1 Simple Program

Let's start with a basic Feynstein program.

```
ShapeScene {
   shapes {
     shape Sphere(name="sphere1", radius=20cm,
       location=(0,0,0));
   }
 }
```

This is as basic a program you can have in Feynstein; a single object with no forces acting on it. This will render a scene to a window on your screen; this window will allow you to pan and tilt using your mouse and keyboard, just like most popular 3D editing software. There are a few things to note, just from this short code example:

1. The source file has a `.f` extension. This is necessary for the compiler and interpreter to compile and run your Feynstein source file.

2. The scene has a name: `ShapeScene`. This name needs to match the name of the source file, and it is the name you will pass in to the interpreter to render your scene.

3. There is block called `shapes`. Within this block, the user can define the shapes she would like to place in the scene. Anything goes in this block, though; anything not preceded by the `shape` keyword will be treated as standard Java code.

4. There is a `shape` keyword. The `shape` keyword defines a type in Feynstein – any kind of shape you add to the scene inherits from `shape`. In the `shapes` block, any `shape` defined on a line by itself is automatically added to the scene.

There is also a syntactical oddity here, as well. Most Java users will be right at home using Feynstein, as almost all of the syntax is borrowed from Java. However, one large irritant of defining a 3D scene in Java is just how many parameters you need to specify for every object. Each object needs an identity, size and location at the minimum, and to make matters worse, both the size and location of every object have dimensions. Feynstein adds a new way to create objects, called âĂIJBuilder SyntaxâĂİ, which lets you use key-value pairs to initialize an object. You can see this in `ShapeScene.f`; we create a new sphere by calling `shape Sphere(name=âĂİsphere1âĂİ, radius=20cm, location=(0,0));`. Builder syntax allows you to set only the properties you want, allowing the rest to remain at their default value. In addition, Feynstein

Shape	Parameters
RectangularPrism	height, width, location, mass, name
Cylinder	height, radius1, radius2 (optional), location, mass, name
Sphere	radius, location, mass, name
Tetrahedron	edges, location, mass, name
Plane	normal, location, mass, name
CustomObj	file, location, mass, name

Table 1: Built-in shapes and their supported parameters

recognizes most standard units for length, mass, force, velocity and acceleration; commonly used S.I. (meter, centimeter, gram, Newton, etc) and Imperial units (foot, inch, mile, pound) are all supported. For a full list of supported units, consult the Feynstein User Manual.

There are many shapes a user can add that come built into Feynstein, and it's also really easy to import a 3D model from somewhere else. The included shapes are shown in the table below. Note that every shape has the parameters `location` and `mass`.

You can also import objects created in other 3D programs into Feynstein if they're in a .OBJ format. Adding OBJ files is easy; you just add a `CustomObj` shape in the same way you'd add any shape, and you pass the file name of model as the `file` parameter. Feynstein imports it for you, and you can set the `location` and `mass` properties of the shape just like you would for any object.

2.2 Forces

No simulation package would be complete without a full array of forces at its users' disposal, and Feynstein is no exception to this rule. With eight built-in forces, the user can specify almost any real-world scenario by just specifying parameters. An example program that includes forces is shown below.

```
ForceScene {
   shapes {
     shape Sphere(name="sphere1", radius=20cm,
       location=(0,0,0));
   }

    forces {
      force GravityForce(gx=0, gy=0, gz=-9.81);
      force SpringForce(restLength=1m, k=1,
        actsOn=#sphere1, fixedAt=(0, 0, 2m));
    }
 }
```

This scene defines two forces using the same builder syntax that we used to define shapes earlier. We place the same sphere that we had earlier in our scene, but then we attach it to a spring. This simulation would result in a

video of a sphere bouncing up and down before it eventually comes to rest about one meter above where it began. Each type of force has a different set of parameters; in this scene we've used GravityForce (which allows you to configure the direction and strength with which the gravitational force acts) and SpringForce (which allows you to configure the rest length, spring force, the location of its fixed end, if any, and the objects upon which it acts).

These two forces are the most intuitive, but there are many forces you might be interested in using. In addition to GravityForce and SpringForce, Feynstein has the following forces at your disposal. Note that for all forces, the parameter `actsOn` specifies an object (or objects) which that force acts upon.

DampingForce A DampingForce is a frictional force which resists motion, and is a function of mass. You can attach a DampingForce to any object, and that object's motion will be resisted. The magnitude of its resistance is given by a single parameter, `lambda`.

RodBendingForce A RodBendingForce acts upon three particles, which together act like a hinge, to resist the bending of these particles. It has three parameters: the rest length between the first pair of particles `restLength1`, the rest length between the second pair of particles `restLength2`, and the rest angle `theta`.

TriangleForce This is a force that acts against the deformation of a triangle. Given the side lengths of the triangle, the stiffness of the resistance `stiffness`, and the compression-to-expansion ratio `poisson`, it will act against any force that attempts to skew the triangle in any way.

SurfaceBendingForce A SurfaceBendingForce is a constraint force that resists the bending of a four-particle surface along its diagonal. These four particles are arranged in two triangles which share an edge. The strength of the force is a function of the angle between these two triangles, and is parameterized by the resistance to bending, `stiffness`, rest angle of the two triangles, `theta`, and the shape of both triangles.

ContactForce A ContactForce is derived from a SpringForce and acts upon two triangles to resist a collision between them. An equal and opposite force is applied to both triangles which pushes them apart, which is analogous to a collision. A contact force is configured by a spring force stiffness `stiffness` and a minimum distance (`minDist`) that must be maintained between the colliding pairs.

If none of these forces satisfy your simulation needs, it's easy to define your own, as well! We'll discuss how to define your own forces in the Advanced Topics subsection.

2.3 Frame Updates

While you're rendering your scene, you get qualitative output in the preview window. If you're modelling a scientific experiment, though, you'll often need much more precision than that; what if you want to know exactly where an object is in a given frame? That's where frame update methods come in.

The frame update method of a scene is a block that is executed every time the scene is stepped forward in time, but before it is rendered. This gives the programmer an opportunity to observe or change some of the properties of the objects in the scene before the next frame is generated. Let's take the ball-on-a-spring example again to see how this might be useful.

```
import java.io.*;

ForceScene {
    FileWriter writer;

    static {
      try {
        writer = new FileWriter(âĂIJoutput.csvâĂİ);
      } catch (IOException) {
        // handle error
      }
    }

    shapes {
      shape Sphere(name="sphere1", radius=20cm, location=(0,0,0)
          );
    }

    forces {
      force GravityForce(gx=0, gy=0, gz=-9.81);
      force SpringForce(restLength=1m, k=1, actsOn=#sphere1,
          fixedAt=(0, 0, 2m));
    }

    onFrame {
      writer.write(String.format(âĂIJ\%d,\%d\nâĂİ, time, #sphere1
          .getZ()));
    }
 }
```

This code generates the same visualization we had before – a sphere bobbing on a spring – but this time it does something extra; for every frame that is rendered, it also writes a record to a CSV file with the time that has elapsed and the height of the sphere. If we then wanted to go away and do further processing on that data in Matlab or Mathematica, it's in easy, machine-readable format.

There's lots of new syntax and features in this snippet, so we'll start at the top. The first thing you'll notice is the `import` statement – imports work exactly the same in Feynstein as they do in Java, and you have the full Java standard

library at your disposal. You can also see we've defined a scene-wide variable, outside the scope of any of our blocks. Anything that isn't in a predefined block is treated as if it were in a standard Java class. In this case, that means our `FileWriter` is an instance variable of our scene.

There's also two new blocks: `static` and `onFrame`. `static` is called after the system has been initialized and properties have been set set, but before any shapes or forces are created. This is a good time to read input from the user, if any, or to initialize instance variables you want to use in other blocks. That's exactly what we do here, initializing our `FileWriter` so we can write to it during `onFrame`.

`onFrame` is the frame update method we discussed in the introduction to this subsection – it is the method that gets run after all of the forces are applied to the shapes in the scene, but before the scene is rendered. Here, you can set or read any of the properties of a shape; in our case, we're accessing the height of our sphere in the scene. Within the onFrame block, you have access to a special variable, `time`, which represents the number of milliseconds that have passed in scene time.

To get the properties of an object, you have to have a reference to the object itself, but since we created the object in the shapes block, we didn't store a reference to it in a variable. Instead, we access it using its name with the #-operator. We then can treat it like a regular object and call instance methods on it, like `getZ()`.

2.4 Integration Methods and Error Bounds

Unfortunately for the users of any simulation system, physical simulation isn't perfect. While, given the right parameters, it can approximate the real world extraordinarily well, sometimes it fails to do so quite dramatically. The most common source of failure in physical simulation is the systematic error (or "instability") inherent in time stepping.

Time stepping is the process by which the simulator figures out the location and momentum of objects in the next frame, based on the forces acting on them and their location and momentum in the current frame. The simulator uses various numerical methods for solving large systems of equations to project the objects into the next frame, and each of these comes with its own drawbacks. As a user of Feynstein, you get to choose which time stepping methods (also called an âĂIJIntegration MethodâĂİ) you would like to use to render your scene. You can also place limits on the error in the scene, so if the error ever passes a certain threshold, rendering is halted and the user is shown an error message.

Feynstein comes with two different time-stepping methods, each with advantages and disadvantages. These are all properties; for example, if I wanted to use Velocity Verlet with a step size of 2 milliseconds, I would add `property`

`VelocityVerlet(stepSize=2ms);` to my `properties` block. It is important to note that the stability of any method is dependent upon the step size used; a larger step size means lower stability, and vice-versa.

SemiImplicitEuler The semi-implicit Euler method of time integration is, as the name suggests, a midpoint between explicit and implicit. Unlike implicit Euler, it uses the location in the previous frame to calculate the velocity in the next frame, giving it a source of error (especially when you have fast-moving things in your scene). However, it then uses its estimate for the velocity in the next frame to calculate its next position, unlike explicit Euler, which only uses its knowledge of the current frame. This is a good trade-off between speed and stability, and is a good choice for everyday rendering tasks.

VelocityVerlet Although the semi-implicit Euler and velocity Verlet techniques are similar, velocity Verlet is more accurate over long periods of time. Verlet integration updates velocities in two stages, and uses the intermediate velocity to calculate new particle positions.

2.5 Collision Handling

Collision handling in implemented in Feynstein in through two discrete modules: detection and response. You can add a collision detector to your scene without necessarily having a collision responder, but all collision responders need a detector to find collisions to respond to. Collision handling doesn't need to be tied to specific objects – detectors and responders will check all objects in a scene if they are defined.

2.5.1 Collision Detection

Feynstein supports two predefined options for collision detection: proximity-based detection with `ProximityDetector` and a more thorough, continuous method, `ContinuousTimeDetector`, which uses time steps to check for collisions. Both of these methods extend the abstract `NarrowPhaseDetector` class because both are part of the narrow phase of collision detection, where all possible collisions between objects in a scene are monitored.

Broad-phase collision detection constitutes a helpful, but not mandatory, optimization for narrow-phase detection: broad-phase detection uses a `BoundingVolumeHierarchy` to store probable collisions based on a VolumeHierarchy heuristic. To use this optimization method, add a `BoundingVolumeHierarchy` property to your properties list, before you specify a `ProximityDetector` property. Note that just a `BoundingVolumeHierarchy` isn't enough to detect collisions: you also need to have a narrow-phase detector.

```
        properties {
```

```
        property BoundingVolumeHierarchy(margin=0.1,
               type=BoundingVolumeHierarchy.AABB);
        property ProximityDetector(proximity=0.1);
      }
```

2.5.2 Collision Response

Feynstein offers two predefined methods of collision response, extending the abstract `CollisionResponder`: the `SpringPenaltyResponder`, which uses spring forces of a high strength to model elastic collisions between shapes in a scene, and the `ImpulseResponder`, which applies small impulses to neighboring objects to prevent them from colliding. Both responders must have a `CollisionDetector` to function properly; failing to define one will create a compile-time error. Detectors are indexed by the order in which they appear in your properties block, starting with 0, and this integer index is the value for the `detector` parameter in both `CollisionResponders`.

The `SpringPenaltyResponder` takes a proximity (how close objects must be before it responds to a collision) and a strength for the spring force as parameters, while the `ImpulseResponder` takes only a number of iterations over which to apply impulses to colliding objects. All of these parameters have default values, so if you're not sure how precise you want your collision responders to be, specifying only a detector in your builder syntax will cause Feynstein to operate with the values we've defined.

```
  CollisionScene {
      shapes {
        shape Sphere(name="sphere1", radius=20cm, location=(0,0,0)
            );
      }

      forces {
        force GravityForce(gx=0, gy=0, gz=-9.81);
        force SpringForce(restLength=1m, k=1, actsOn=#sphere1,
            fixedAt=(0, 0, 2m));
      }

      properties {
        property SemiImplicitEuler(stepSize=0.01);
        property BoundingVolumeHierarchy(margin=0.1, type=
            BoundingVolumeHierarchy.AABB);
        property ProximityDetector(proximity=0.1);
        property SpringPenaltyResponder(detector=0, proximity=0.1,
             stiffness=1000)
  }
```

3 Project Plan

Colleen McKenzie, Project Manager

3.1 Development Process

Two aspects of our development process that I believe contributed to the project's success were the modularity of our language and compiler and our decision to build our language and our code base in increments. There were, of course, general aspects of the language that had to be decided on early in the project, such as the overarching structure of a Feynstein file, and these were discussed and voted on at group meetings. Our Language Guru defined a hierarchy of discrete features for inclusion, so that we could start out with a small but functional code base and build on it to include as many possibilities as possible for our languages users. (See section on Language Evolution for an in-depth description of this process.) Because we divided our language into sections in accordance with physical elements–there are shapes (masses), forces, and properties such as time integration and collision handling–the modules of our translator could be designed, developed, and tested independently of each other, which decreased the dependencies in our language and allowed the maximum flexibility for group members to work at their own paces to finish the sections of the project assigned to them.

3.2 Roles and Responsibilities

While our language was modular and easily separated into discrete sections, the responsibilities of the project's group members were far less distinct than we expected and than the course materials' guidlines suggested they be. I believe this unsystematic distribution of work speaks to the flexibility of our project group, but certain areas of the project may have run more smoothly if we had made firm decisions about what our group roles constituted early in the course of the project.

Colleen McKenzie: Project Manager I was responsible for organizing meetings and taking minutes at group meetings. I also handled updating the group's project wiki and Google Calendar with minutes, resources, events and deadlines, and other changing information pertaining to our language and its implementation. Monitoring group members' progress was also my responsibility, and I regularly checked in with group members to keep track of progress in each contributor's area of language development and implementation. This monitoring also included the setting (and resetting) of deadlines and querying group members about their progress. Finally, I was responsible for proofreading all documentation (typeset by our Systems Architect) and either correcting errors or communicating issues to the Language Guru.

Samantha Ainsley: Language Guru Samantha was responsible for providing the conceptual organization of the language–she was our essentially our

Physics Guru, and she used her expertise to lead us to consistent and logical decisions about the design of our language. She provided reference materials to give us an idea of how other languages and libraries implemented functionality similar to Feynstein's and was the designated point person for questions related to such matters. Further, Sam was largely responsible for executive decisions about which features our language would and wouldn't support.

Will Brown: Systems Architect Will was responsible for defining the structure of the Feynstein compiler code package, which is broken up into modules for translation and for processing different simulation aspects. In addition, Will was entirely responsible for the translation portion of the compiler: he worked with our Language Guru's design to ensure that our language was properly translated into Java code built on the compiler architecture he designed. Will also managed the repository for the group's code and managed the documentation process: he was responsible for compiling and typesetting all of our group's written material.

Rob Post: Systems Integrator Rob was responsible for researching information we used to design and add features to our rendering module, and make decisions about what features and options for visual output we would provide users of our language. Rob also collaborated with the Systems Architect to implement robust compile-time error handling in our translator.

Eva Snow: Tester and Validator Eva was responsible for implementing unit tests designed to make sure our language properly supported all the features defined in our manual and ensuring that the physical features of our language had been implemented correctly.

3.3 Implementation Stylesheet

Our group did not have an implementation stylesheet. Our development process included group members reading through project code before contributing more, and because our Language Guru and Systems Architect defined a robust API while laying the groundwork for the compiler, members contributing new modules were able to sufficiently match the project's existing code.

3.4 Project Timeline

1/24 We decided on and assigned roles for each team member, made logistical decisions (e.g. weekly meetings), and decided on physical simulation as the domain for our language.

1/30 Group members gained familiarity with the language domain, including higher-level physical concepts we would need to implement.

2/6 We developed and recorded milestones for our project and started working on a general language specification to guide future language developments and additions.

2/13 With a few tweaks, we made our earlier language specification final. By this point our builder syntax is well-defined.

2/20 Our language whitepaper is finished at this point.

2/27 We defined more specific milestones for implementing the compiler, and we identified and accounted for dependencies in the structure of the language and the compiler.

3/29 By this point, the language reference manual and tutorial were complete. We had a solid specification and a grammar for our language, both of which were very close to the grammar of our finalized language.

4/3 Translation for Feynstein-specific syntax was implemented, and early translator testing was implemented alongside the translator. Our git repository was up and running, and all members were able to commit code. We implemented basic functionality of our renderer so that it could could display a basic shape to the screen. Mouse and keyboard commands were implemented with the rendered so that users could change the view of their scene.

4/10 Automated compilation and running of Feynstein files was implemented by this point. The modules in our compiler were divided up discretely among group members.

4/18 Time-stepping was implemented, but not integrated into Feynstein yet.

4/24 We implemented basic discretization of geometry: at this point we had fully functional objects in our language, and these could be operated on by force constructs. Basic forces (gravity, springs) were also implemented.

5/3 All forces were implemented by this point.

5/5 Basic force testing was completed and collision detection was implemented by this point.

5/6 Collision response was implemented, and the translator was extended to handle errors in more specific and descriptive ways.

3.5 Project Log

Please see Appendix **??** for the (rather lengthy) project log.

4 Language Evolution

Samantha Ainsley, Language Guru

The Feynstein language evolved rather smoothly throughout the course of this project. From the very beginning we agreed upon a syntax and feature set for our language. We defined a set of clear milestones that we then organized into a language pipeline. Naturally, our first goal was to have our parser and syntax interpreter up and running, so our execution code could be written with few changes to the translator and interpreter. After this, our second milestone was to implement our renderer, without which no testing could be performed as Feynstein requires visual display. Both of these hurdles were overcome in the first half of the semester. With this foundation established, we were able to divide our language into independent features and assign development accordingly.

The dependency graph shown in Figure Figure 1 shows the work flow for our language development. An arrow indicates a dependency, i.e. if A->B, then A must be completed before B. Features shown in italics were deemed second priority because no other language modules depend upon them and they are not necessary for testing, or there exists an equivalent feature. Based on this flow graph, we decided to assign integration and shapes to different team members as they could clearly be completed in parallel. The team member responsible for integration would first complete one integrator before developing an additional integrator, so the person responsible for forces could begin their work. The person responsible for forces would first complete spring forces, as collision response will depend on this logic (and it is the simplest force potential). Forces where An additional person would then be responsible for handling collisions, which we agreed upon as a second completion state. As collision detection extends all other primary modules, it could be left for future development with the language in a completed state under undesirable circumstances.

Our syntax, which is intentionally close to Java for easy integration of Java code into Feynstein programs, improves upon Java with a feature we called Builder Syntax. Builder Syntax allows users to specify Feynstein object instantiation parameters in an arbitrary order with key-value pairs. For example, a user creates a gravity force as `force GravityForce(gy=-9.8)`. The GravityForce takes additional optional parameters gx, and gz, which specify the acceleration due to gravity in the x and z directions, respectively. This important feature makes Feynstein accessible to users without requiring an in-depth understanding of its API.

The systems architect took primary responsibility for developing the translator logic of Builder Syntax translator was completed in the first half of the project timeline. Meanwhile, the language guru wrote the OpenGL pipeline

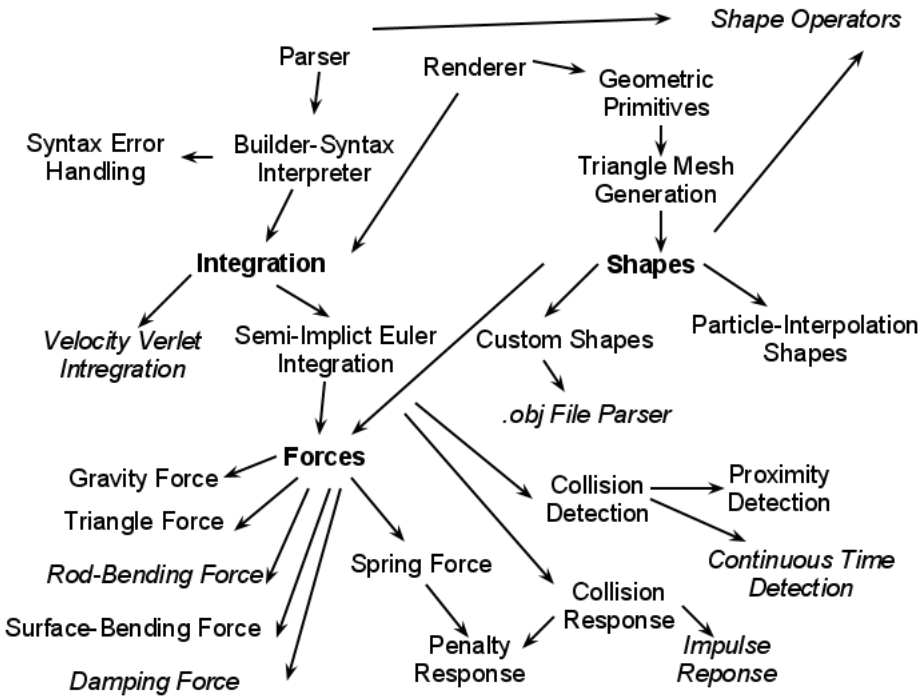

Figure 1: Dependencies in Feynstein

render as a stand-alone Java application to be incorporated into the make Feynstein program. The architect's Builder Syntax implementation allowed for each development of new-key value pairs without modifying the basic grammar. Instantiation parameters are configured as a chain of function calls, each of which returns the updated object instance. The syntax was also developed to support list configuration when setting multidimensional parameters; for example, `shape Sphere(position=(0.0,0.2,0.2))`. Builder Syntax keys translate to methods of name `set_<key_name>` that return an instance of the class type–this facilitated rich feature development for the rest of the team.

With the goal of allowing the team to easily develop properties, shapes, and forces independently, our architect wrote the APIs for each base class. With our code checked into a central repository, the other team members could fill in the method logic accordingly. Geometric primitives were developed for the renderer and used to build a triangle mesh class for shape topology. As planned, we developed our initial custom shapes in parallel with the Semi-implict Euler integrator. We implemented the .obj file parser first, as this would allow for quick render testing without defining complex geometric configurations dynamically. We then developed forces while extending our custom shape library.

Forces were developed in order of increasing complexity measured in the number of particles they act upon. The most complex force we aspired to support, a tetrahedral constraint force, was omitted from the language as it only applied to a single shape configuration. We developed particle-interpolation shapes, which are small shapes defined by a list of vertices, in light of forces for easy unit testing of force stencils.

We had hoped to develop a second time integration class in parallel to force and shape development, but did not reach this goal. The ImplicitEuler method we hoped to support required a complex multi-dimensional numerical solver for which we could not find an appropriate external library. We did however, add a less complex Velocity-Verlet based integrator to offer users more flexibility. We completed all force and shape definitions soon enough that we could focus our last efforts towards building our collision detection system while simultaneously developing a syntactical feature that allowed users to manipulate shapes with predefined operators. We first completed proximity-based collision detection, then continuous-time collision detection, which is significantly more complicated and precise. We then developed a penalty-spring-based collision handler followed by an impulse-based handler.

With the exception of our two minor changes, our language evolved beyond our first completion stage. We achieve our milestones on time, though sacrifices in division of labor often had to be made to reach this goal. With better deadline achievement, I believe we could have achieved more interesting rendering capabilities into the languages–such as lighting and textures. Nevertheless, our final feature set provides a rich framework for configuring physics-based simulation as well as many easily-extendable building blocks for future research and development.

5 Language Architecture

William Brown, System Architect

The Feynstein system is broken into two distinct portions; the translator and the Feynstein execution environment. The translator takes, as input, Feynstein source code and translates it into valid Java source code. This Java code is then run in the Feynstein execution environment, which pulls in all of the Java code we have written to provide the rendering, geometry and simulation constructs that are present in Feynstein.

5.1 Translator

The translator uses a multi-pass system to translate Feynstein into Java. It first does basic lexing, in which it breaks inputs first into statements, then

into the lexemes within those statements. After lexing, it constructs a syntax tree. Once the code exists within this tree, the translator walks the tree to generate valid Java code. In most cases, this is simple textual substitution; for example, the Feynstein code segment `#myShape` translates directly into `getShape(‘‘myShape’’)`. However, some constructs require a more in-depth parsing; in particular, builder syntax was quite difficult to translate. These statements, of the form `MyObject(attribute1=value, attribute2=value)`, were not valid Java and thus didn't cause a collision in our grammar, but still were difficult to distinguish from, say, `MyObject(x == y)`, which is valid Java (if `MyObject` takes a boolean as an argument to its constructor).

We do have one unusual step in our translator, which is "structure analysis" – since Feynstein has certain parts of the language which this user is required to use (for example, the user must specify a "shapes" block), we analyze the syntax tree to ensure that they've incorporated the required portions of the source code.

Error handling in Feynstein is difficult; we need to handle errors returned by the Java compiler and virtual machine and give users some indication as to where in their original Feynstein (instead of the generated Java code) the error occured. To do so, during parsing and syntax analysis we generate a mapping of Java line numbers to Feynstein line numbers. Then, if the Java compiler or VM throws an error, we mine the stack trace for the Java line number which causes the error and tell the user which line number in Feynstein generated that Java code.

Both Rob and Will contributed to the translator. Will wrote the initial implementation of the translator and was responsible for implementing new features at the translator level. Rob was responsible for error handling.

A block diagram of the translator is shown in Figure 2

5.2 Execution Environment

The execution environment of Feynstein is the more complex half of the system; while our language is very similar to Java and therefore requires simple translations, the actual behavior of a Feynstein program is extremely complex. This complexity is mirrored in the complexity of the architecture in the two portions; while the translator had four modules and a fairly simple API, the execution environment is composed of almost 7000 lines of Java code split across 50 classes. A block diagram of the execution environment is shown in Figure 3.

Will defined the API for the components in the system to ensure consistency and compatibility with translator output. From a high-level implementation perspective, Sam was responsible for rendereing, geometry, forces, and time integration, Colleen was responsible for collision detection and response, and Will was responsible for shapes. For a more in-depth summary of who was

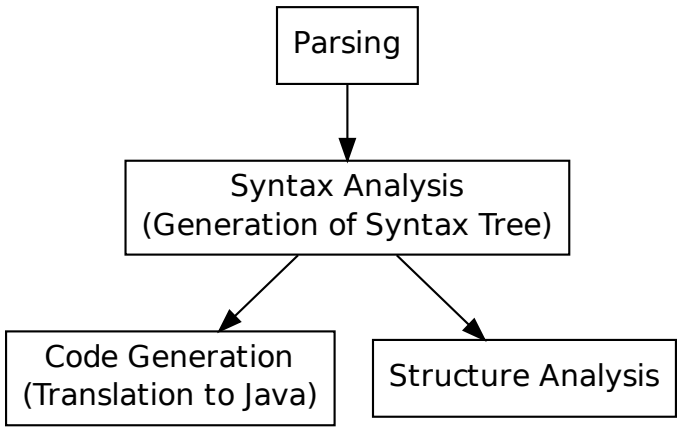

Figure 2: Block diagram of the stages in the translator

responsible for what, see Appendix **??**, and for a line-by-line listing of who wrote what code, see Appendix **??**.

6 Development Tools

William Brown, System Architect

6.1 Source Code Management

In order to manage our team's source code, we used Git, which is a distributed version control system. Will set up a Git repository on his server, to which everyone on the team was given SSH access. It allowed all of us to simultaneously work on Feynstein and then easily merge our changes together. Git also allowed us to easily track peoples' contributions to the code base, as it allows a user to see exactly who wrote each line of code in a repository. Will wrote a shell script that allowed us to see exactly who wrote how much, as well, which helped for the final calculations to ensure that everyone had written at least 500 lines of code.

6.2 Languages, Tools and Libraries Used

We wrote our execution environment in Java, as this was the only language with which the entire team was familiar. It also provided us with the portability inherent in Java, which was important for a team of two Mac users, a Windows user and two Linux users.

We make heavy use of OpenGL for our graphics, but there isn't a native OpenGL implementation in Java, as it requires many "bare-metal" computations that aren't possible in a virtual machine. Instead, we used Java OpenGL[1] (or "JOGL"), which uses JNI to call native C libraries on the machine. JNI is notorious for being difficult to set up, and JOGL lived up to this expectation; while we found it was easy to install for the members of our team who use Linux, it was extremely difficult to install on OSX.

However, the installation procedure was worth it; JOGL gave us the full power of OpenGL, and allowed us to use OpenGL at near-native speeds and efficiency. Without it, this project would not have been possible.

On the translator side, we used PLY[2] (Python Lex-Yacc) for lexing and parsing. We wanted to use an implementation of Yacc in a scripting language, as most of the actions in our parser were string manipulations, and string processing tends to be easier in scripting languages. We used Python as this is what Will and Rob were more familiar with.

7 Testing Plan

Eva Asplund, System Tester

Because there are so many components to Feynstein above and beyond merely translating the code, much of the testing methodology was geared towards validating the implementation of the physics of the program. Testing for these elements involves rendering actual scenes written in Feynstein in order to visually verify that the math in the code is working. The Feynstein tests for built-in forces were modelled after XML files used by our Language Guru in her physical simulation work. As new forces and physics functions are added to the program, the suite of visually-verifiable test programs can expand accordingly, and is re-run in order to assure that the new code has not caused an unintended change in the functioning of the old.

The Feynstein translator test consists of a Feynstein source code and corresponding output in java. By running the code through a new version of the translator and saving it before it is compiled and run, one can verify that the new translator performs equivalent transformation. Once this new output is

[1] `http://jogamp.org`
[2] http://www.dabeaz.com/ply/

compiled and executed to ensure its validity, it can be saved as the test for the next generation Feynstein translator.

8 Conclusion

8.1 Team Lessons

We discovered the importance for having a central repository for all updates to our grammar, of which there were many and which were often communicated between members of the group instead of being recorded.

We learned the importance of defining general responsibilities in advance: our roles were far less discrete than we expected them to be, and while this is indicative of the cooperative way we approached this project, it led to an unequal division of work that could have possibly been avoided.

We learned that the more modular our language was, the easier it was to add–and remove–language components easily.

8.2 Individual Lessons

Colleen: Project Manager My belief in the validity of Hofstadter's Law–"It always takes longer than you expect, even when you take into account Hofstadter's Law"–has been confirmed many times over in the course of this project. I've learned a lot about the importance of setting realistic deadlines, but I've realized that it's equally important to accept that sometimes a schedule you thought was realistic at first needs to be radically changed.

On a more individual note: I also learned the importance of understanding different styles of communication and of figuring out how one's team members work best: I believe that a few pitfalls could have been avoided if I had worked on our group's communication practices earlier in the project timeline. I initially approached the project as a very lenient manager, assuming that my group members–myself included–would produce work perfectly and according to schedule. I learned quickly that expecting people to behave as predictably as machines was unrealistic, and I began to understand the importance of instating policies for unfavorable situations at the beginning of the project, rather than once those situations have already happened.

For our group, these situations mostly involved missed deadlines and an ensuing redistribution of work to the group members most able and willing to complete it. I began the project with the goal of giving all group members considerable enough contributions that they would all feel a sense of ownership for the project, but ultimately this was now

how our group functioned, and I learned that at a certain point it's necessary to assign work to whoever will get it done quickly. A closely related lesson–and the hardest one for me to accept–was that sometimes the Project Manager has to prioritize completing a project over being nice or trying to keep everyone happy.

Sam: Language Guru Working on this language was fascinating for me, especially in a group dynamic. I proposed this language idea to my teammates because physical simulation is my primary area of research. The idea was very much inspired by my own experience learning simulation. I took a course in physical simulation that was taught entirely in C++, a language I was unfamiliar with at the time. I found (and I was far from alone) that most of my energy was put towards learning to use a new language in a very complex way: building an interactive object-oriented system and optimizing for performance. Although this was a great learning experience, within that particular course, I wished I could have focused more on understanding the physics behind the engine I was writing rather than just getting it to work–so much time was spent on the initial implementation that little was left for experimentation. As a result, I was inspired to create a language that facilitated easy understanding of the physical simulation concepts, but also allowed for more complex extension and experimentation.

Needless to say, there was great pedagogical merit in this process. I am interesting in learning how different people understand the task of programming physical systems, and this gave me great insight. The language design is very much a reflection of this pedagogical experience. The act of teaching new concepts involves abstracting the lower level details into comprehensible objects and relationships. That said, my initial lessons for teaching simulation to the team ended up defining our language model. Furthermore, by working in Java, I learned a great deal about adapting my own perspective of developing simulation code, which is characterized by C++, to new languages–especially in terms of achieving optimal performance.

Simulation aside, another important lesson I learned is about the dynamics of team work. In past experience, I have always tried to divide up labor as efficiently as possible so the maximal number of components could be developed in parallel. This division of labor was always straightforward when each person considered themselves a unique specialist. But what if not everyone is a specialist? I our group, there were some members who came in knowing exactly what they wanted to do, and others who were more flexible. There is, of course, an appeal to the person who is willing to work on anything, but from what I have learned, if someone considers themselves an expert, they will perform accordingly. I think

that passion makes a team, and team members need to really to be confident in their necessity within the larger group from the beginning. I've learned that when selling an individual vision to a group of teammates, you should be certain each person is excited about their role.

On that same token, one of the hardest parts of the language guru's job is letting go of control. When you envision a perfect system, you can't imagine accepting outside input. In some cases, my teammates showed me their clever and new perspectives on methods with which I've become rather familiar with developing simulators. It was admittedly hard to assign major coding components with which I was so familiar to my teammates. Some team members were very responsive, and I enjoyed collecting relevant reading materials to help them understand larger concepts, helping them through the coding process, and seeing their genuine excitement when things were working. Unfortunately, in other cases I believe some tasks may have been too daunting for team members. In these cases, I learned that you have to be aggressive when offering help–people won't always ask for it. On the other side, I also learned that sometimes you need to be resolute and put the language first, even if that means redistributing the work load towards those who consistently complete their work on time.

Will: System Architect I learned the value of defining APIs early on, because just the definition of an API is enough to get everyone thinking about how exactly things work. We had some issues later in the semester in which things couldn't work as we had planned, and we would have realized that that was the case if we had defined just what methods and fields each class in our execution environment had.

I also learned (from experience) why everyone uses parser generators like YACC; at first, I tried to implement our language in pure Python, starting from scratch. I found that this approach made it much easier to write the actions in the SDTS for our language, but it made it extremely difficult to adapt to changes in our language's grammar.

Finally, I learned a lot about my own division of labor when it comes to working in a team. Specifically, I found that the most challenging part of the project was responding to emails and maintaining the administrative part of the project; the coding came comparatively easily, despite the fact that I did a lot of coding. In fact, the largest headaches came from my job managing the documentation – compiling everyone's contributions, ensuring that things got done on time and responding to people's requests for edits in a timely fashion were probably the most difficult tasks that I had.

Rob: Systems During this project, I learned two things. The first and probably most important thing I learned is that communication is essential in

successful group work. I learned that I do not do this well and that it is something I need to improve upon if I want to be a more successful and productive group member in the future. The second thing I learned is that starting to code early is also really important in getting the best results as possible. It allows time to revise the code and make sure it works properly. That is also something which I will need to work on.

Eva: Testing You never know how much you're going to miss a debugger until it isn't there anymore. It's pretty amazing the amount of knowledge contained in the heads of 5 Computer Science majors, but it's only helpful if you're asking questions. Teammates and classmates can be a great resource, but only when you take the initiative to communicate and ask questions.

8.3 Advice for Future Teams

We found the existence of our team wiki to be very helpful, and would encourage other groups to implement their own and use it even more extensively than our group did. We would suggest that other groups make sure that the basic framework for working together on a coding project is in place early in the project. This framework would include expectations about meetings and communication in addition to tools to store, update, and communicate about the compiler's code at all stages of the project. There are few replacements for effective communication; in some cases, team members missed deadlines just because their teammates didn't respond to emails in a timely manner. Starting early and communicating effectively are both key to meeting deadlines with high-quality results.

We also would advise future teams to set as many bail out points as possible. Define versions 1.0, 1.1, and 1.2 of your language. Aspire for version 1.2, but be willing to accept 1.0. This was definitely the saving grace of our design process: we had many small milestones. Also, it is important to never hesitate to ask team members for help. This was definitely a struggling point for our team. Remember that asking for help isn't a reflection of a lack of intelligence, and more communication between teammates is always a good thing. Getting stuck and not telling anyone is never an excuse for failing to meet a deadline. On that same note: meet and meet often. Lastly, be certain from the very beginning that every team member knows the answer to these three questions: what is their role, why were they chosen for it in particular, and are they excited about it?

8.4 Instructor Feedback

Sam I found the material for this course to be as interesting as it was difficult to grasp. My first suggestion would be more consistent written homework–

however small–to reinforce ideas as they are learned. I would also suggest more consistency between homework and exams. I found that whereas the homework were more procedural, deriving from examples in the book, the exams were more conceptual.

As far as the project is concerned, I would suggest a clearer definition of the five roles from the beginning. I remember our team did not gain a clear understanding of the distinction between the systems architect and integrator until the final report descriptions were posted. On that note, I often found that project-related assignments, such as the white paper and the language manual, did not come with explicit description. Perhaps more lessons focusing on the language design process would be beneficial.

Colleen I found the sections on run-time environments (chapter 7) code generation (chapter 8) to be particularly interesting–both gave me a far greater understand of facets of computer programming that I had previously taken for granted.

As my group's project manager, I also loved Bob Martin's presentation on software projects, but I would have appreciated the information even more early in the semester, before I had already run into some of the problems he warned us about. In general, I would have liked to see more information about project group roles and advice about potential pitfalls early in the semester.

Will I really enjoyed all of the material associated with this course, and I think the project really gave me an appreciation for compilers and compiling tools. If I were to change one thing about this course, it would be to make homeworks more frequent. I would have liked to have more opportunities to apply some of the knowledge that I learned in lecture. Also, I'm not sure if this is unique to our language or all teams' languages, but we never got into any code optimization in Feynstein. I would have found a homework that required me to implement some basic code optimization really challenging and enjoyable.

Eva I think we covered grammars and regular languages a little more extensively than we needed to given that they were review. It would have been nice to get to YACC and lex right away.

Rob For future classes, I'd like to see a bit more examples of other programming languages and syntax styles, for no other reason other than I think things like that and I think it would be interesting and entertaining.

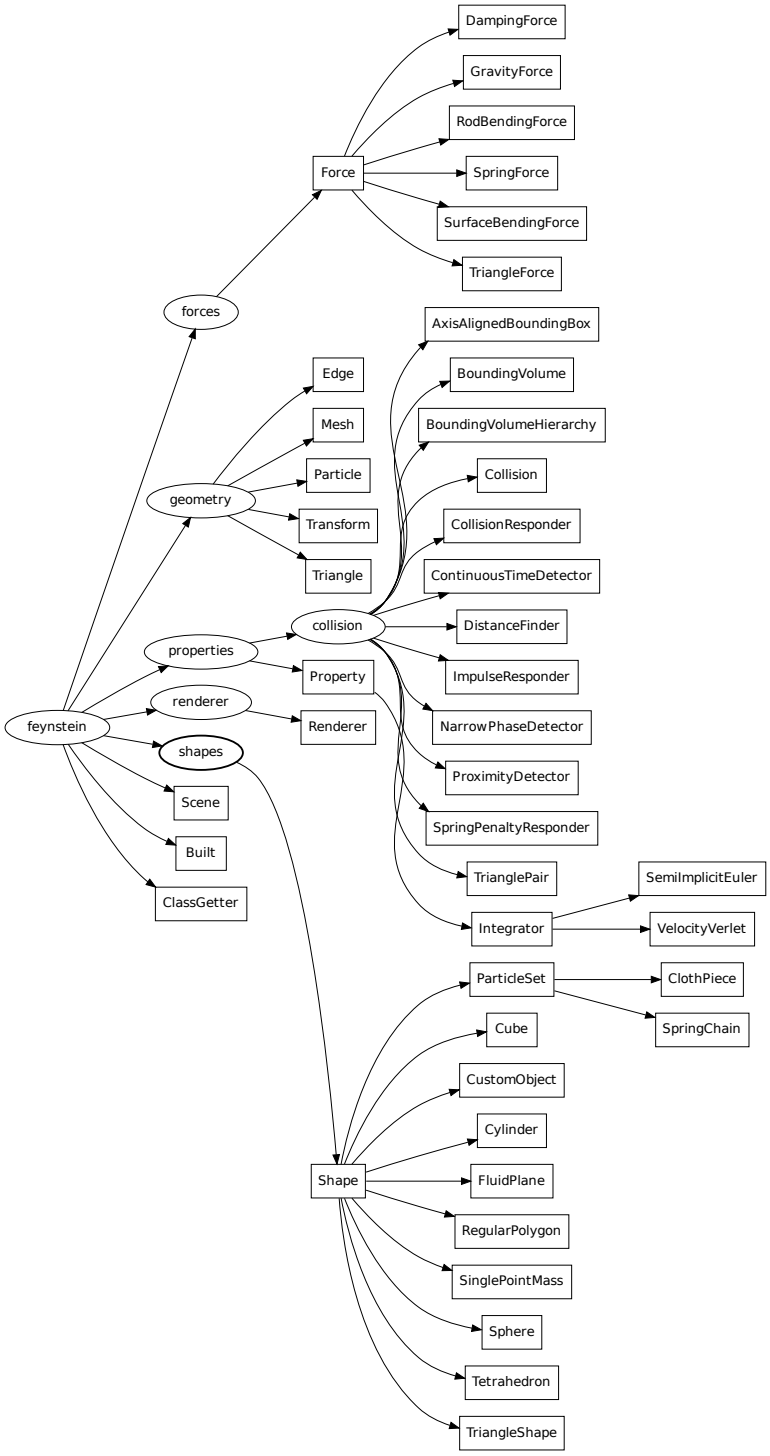

Figure 3: Block diagram of the execution environment architecture