CSS451 Final Project Userguide
Crash! – A Freeform Driving Game
Jordan Phillips

I. Overview……2
II. In-Game Controls……2
III. In-Game Interface……..3
IV. Menus……..4
Appendix A. Using ODE……..6

I. Overview

Crash! is a freeform arcade driving simulation. There are no objectives and no levels; do what you want and go where you will. Crash! makes use of a fully featured 3rd party physics engine (ODE – http://www.ode.org) and is written in C# utilizing the OpenGL graphics API.

There are seven drivable vehicles, and one non-player vehicle. Drivable vehicles are; a Delorean, a Formula 1 race car, a H2 (civilian Humvee), a military Humvee, a very small civilian car, a jeep, and a stock car. The one non-drivable vehicle is the police car – but be on the lookout because breaking the speed limit is a very serious crime.

II. In-Game Controls

The infamous WSAD control scheme is used to drive the player's vehicle. The "W" key accelerates the car forward, the "S" key accelerates the car in the reverse direction, the "A" key turns the front wheels left, and the "D" key turns the front wheels right. In addition to the movement keys, the Space Bar applies the handbrake (only while the key is depressed).

The "C" key cycles through the available cameras – which are:

- **Behind**
  Shows the player vehicle from slightly behind and above.
- **Far Behind**
  Shows the player vehicle from far behind and slightly above.
- **Front**
  Shows the player vehicle from the front and above.
- **Overhead**
  Shows the player vehicle from directly overhead. This camera's up direction matches the forward direction on the player vehicle.
- **Free**
  This camera does not follow the player vehicle in any way. When this camera is selected, the user may rotate the camera around the focus point by holding down the left mouse button and moving the mouse. Holding down the right mouse button and moving the mouse right or left will zoom the camera in and out (respectively), while holding down the middle mouse button (or scroll wheel) will cause the camera's focal point to move.
- **Driver**
  Shows the view from the driver's seat (centered between driver's seat and passenger seat) of the player vehicle.
- **Drive-by / Cinematic**
  This view fixes the camera at a point above the center of the game world, and follows the player vehicle.

The “F” key will reorient the player vehicle. That is, it will right the vehicle so all four wheels are in contact with the ground, and will position it so that it is facing in the original direction. This is very useful if the vehicle has flipped over, or become stuck in some way, and the wheels cannot make contact with the ground or some other surface.

III. In-Game Interface

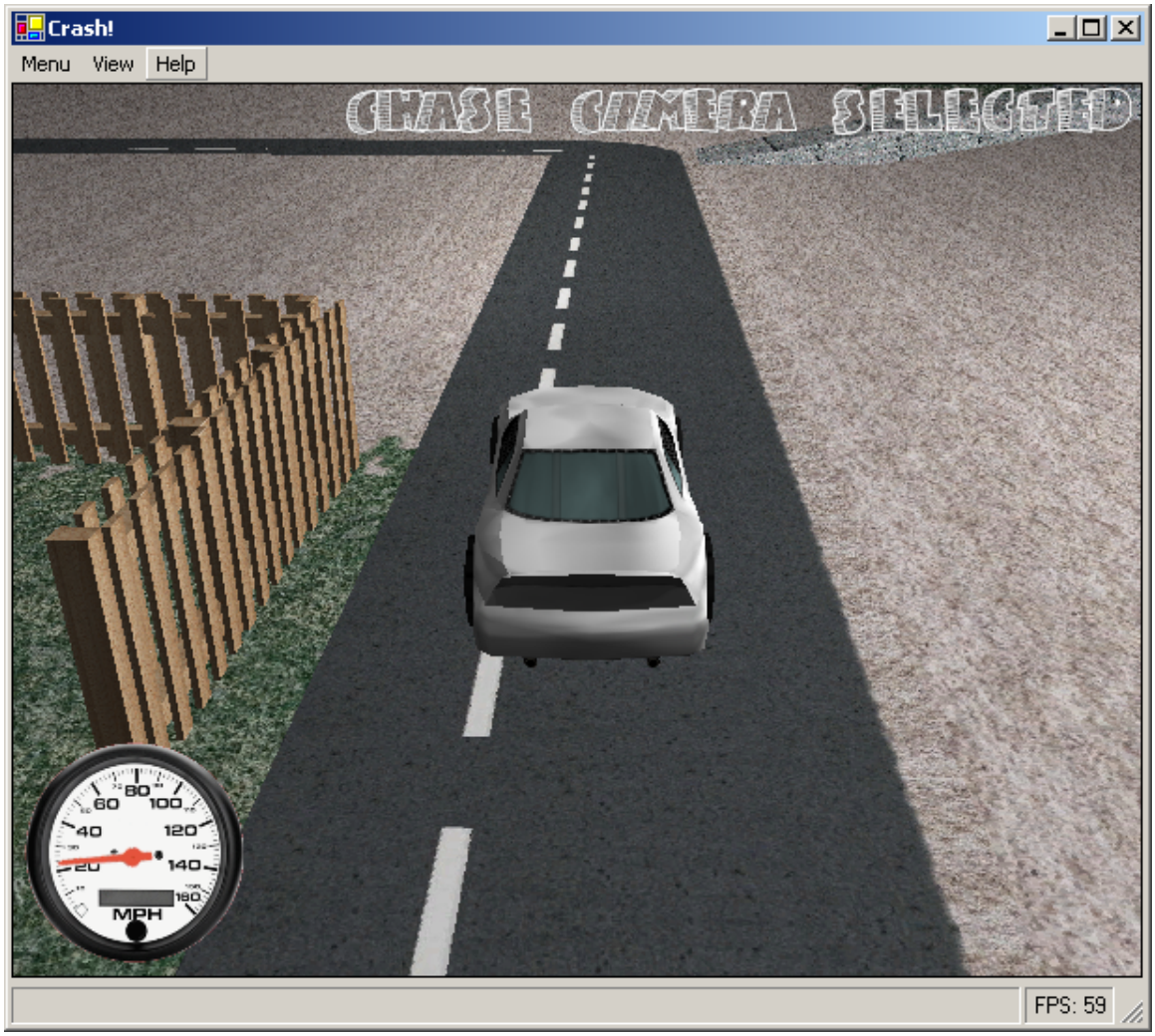

Figure 3.1 – Main Game Screen

Displayed on the main game screen is a speedometer (seen above in *Figure 3.1*). The speedometer shows the current speed as calculated from the rotation and radius of the read tires (this is important to note as applying the handbrake will lock the rear tires, and the speedometer will display zero even if the vehicle itself is still in motion). The speed limit is 40 MPH…

When the camera is changed, the name of the newly selected camera will be displayed in the upper right corner of the game window. Watch that area, as other important messages will be displayed there.

In the bottom right corner of the main window – in the status bar – the current number Frames Per Second that the game is rendering is displayed. Occasional non-game related system messages will be displayed to the left of the FPS display.

IV. Menus

The "Menu" menu (as seen in *Figure 4.1*, below) contains the main game options. The options are:

- **Pause**
  Pauses the game. Checked when the game is paused.
- **Reset**
  Resets the game to the original condition. Note that the currently selected camera will remain as selected and will not be reset to the free camera– this is the only exception.
- **Change Vehicle**
  Opens the change vehicle window, which allows the player vehicle to be changed.
- **Shoot Fireworks**
  When enabled (will be checked when enabled), fireworks are shot from the player vehicle.
- **Enable Frame Limiter**
  The frame limiter locks the frame rate at 60 FPS (or below if the machine cannot keep up). The frame rate limiter is enabled when this item is checked, when disabled, frames are drawn as fast as possible.
- **Quit**
  Exits the program.

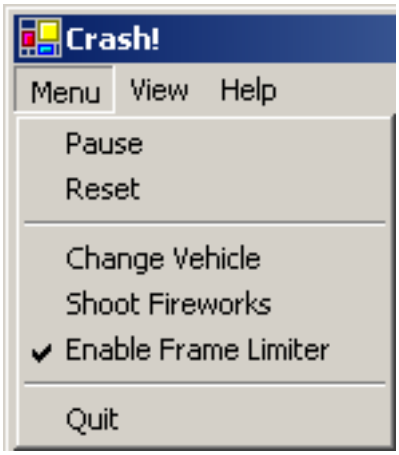

Figure 4.1 – The "Menu" menu.

The "View" menu provides options related to the in-game view. The options are:

- **Camera View**
  Highlighting this option will bring up a list of the different cameras (as listed in section II). Clicking on one of the camera items will change the current camera.
- **Show Speedometer**
  When checked, the speedometer is shown – otherwise the speedometer will be hidden.
- **Overview Window**
  Opens a secondary window which provides a "free" camera, and displays the location and orientation of the normal game view's camera.

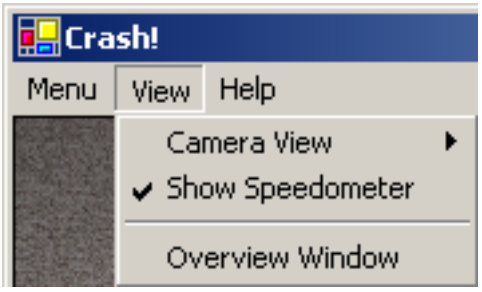

Figure 4.2 – The "View" menu.

The "Select a Vehicle" window allows the player vehicle to be changed. To change the vehicle, simply select the desired vehicle by clicking on its picture, and click the "Okay" button.

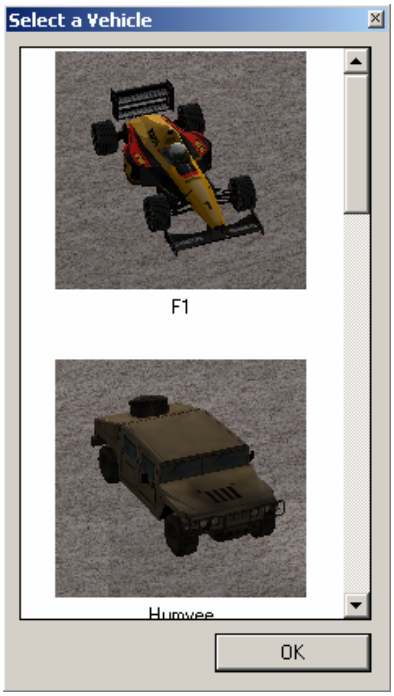

Figure 4.2 – The "Select a Vehicle" window.

Appendix A. Using ODE

Preamble

ODE is the Open Dynamics Engine – an open source physics engine written in C. As it is written in C, it is not object oriented. However there are numerous object oriented wrappers available for it – and in this case, the managed C++ .NET wrapper created by David Walker (http://homepage.ntlworld.com/david.walker530/ode/). Another managed C++.NET wrapper is available which uses a more recent version of ODE (0.5) and takes a slightly different approach (http://www.thejamesrainenetwork.co.uk/ode/ode.html). Finally, if OOP is not your thing, the Tao Framework should soon include a C# wrapper for ODE that is non-object oriented (http://www.taoframework.com/).

Code in this tutorial is C# using David Walker's managed ODE wrapper, but should be understandable as pseudo-code for anyone not familiar with C#.

Who is this for?

This tutorial is meant as a brief introduction to ODE and physics engines in general. There is plenty of information available on getting ODE running (see http://www.ode.org and previously mentioned ODE.NET wrapper links) – as well as on more advanced topics once you have ODE setup and integrated into your application. This tutorial falls somewhere in between.

The idea

The very basic idea of using a physics engine is to remove the burden of complicated physics programming. Instead of implementing collision and movement routines for the objects in our game or simulation, we will create analogous objects in the physics engine, and "animate" our game objects by synchronizing them with their physics engine duplicates.

Bodies and Geometries

ODE simulates two separate entities. The first is a body. A body can have a force – such as gravity – exerted on it, and can have attributes such as mass, position, orientation, etc. Geometries are used purely for collision. Associating a body with a geometry gives you an object that can collide with other objects, as well as be affected by non-collision forces (again, think gravity here).

Worlds and Spaces

When you create a body, you place it in the physics world. This is a representation of some infinite volume in which there can be a universal gravity as well as other (more advanced) universal properties.

When you create a geometry, you place it in a space. This is merely a collection of geometries that can collide with each other. Two spaces may also collide with each other – resulting in collision between all of the geometries that they each contain.

Joints

A joint is a connection between two bodies. As it is related to bodies, not collision, a joint is created in the world. A joint exerts some force on the bodies it connects to maintain a constraint between them. There are a number of different joints, depending on the constraint you wish to have.

Collisions

When a collision occurs, a collision callback function (implemented by the user program) is called. This callback is given the two geometries that are colliding, and generally (for standard collision behavior) should call an actual collision function for the two geometries, and then create a special kind of joint between the bodies associated with the geometries, called a "Contact Joint".

The physics model update loop

A typical loop may look something like this (you are ready for some code, right?):

```
while (something)
{
      // timeElapsed is time since last update
      UpdateWorldForces(timeElapsed);
      ContactGroup.Empty();
      Space.Collide(CollideHandler);
      World.Step(timeElapsed);
}
```

UpdateWorldForces(…) should exert any forces – such as a user pressing the forward button in a driving game – that are external to the physics simulation itself.

ContactGroup is a collection of Contact Joints, which was filled by the collision callback routine in the previous collision step.

CollideHandler is the callback function.
World.Step(…) is the function that actually causes the physics engine to compute an update for the given time step.

The CollideHandler callback function

```
protected virtual void CollideCallback(Ode.Geom o1, Ode.Geom o2)
{
  // If neither of the Goemetries has a body, there can be no
     collision, since there is nothing which can have a force
     exerted on it.
  if (o1.Body == null && o2.Body == null)
    return;

  // If the bodies are already connected by some sort of Joint, we do
     not want to collide them.
  else if (o1.Body != null && o2.Body != null &&
           o1.Body.IsConnectedTo(o2.Body))
    return;
```

```
  Ode.ContactGeom[] cgeoms = o1.Collide(o2, 3);
  if (cgeoms.Length > 0)
  {
    Ode.Contact[] contacts = Ode.Contact.FromContactGeomArray(cgeoms);

    for (int i = 0; i < contacts.Length; i++)
    {
        contacts[i].Surface.mode   = Ode.SurfaceMode.Bounce;
        contacts[i].Surface.mu     = 500;
        contacts[i].Surface.bounce = 0.3f;
        contacts[i].Surface.bounce_vel = 0.2f;

        Ode.ContactJoint cj = new Ode.ContactJoint(World, ContactGroup,
                                                   contacts[i]);
        cj.Attach(contacts[i].Geom.Geom1.Body,
                  contacts[i].Geom.Geom2.Body);
    }
  }
}
```

There is a lot going on here, so let's take a look at some important parts. First, there are a couple conditions we impose on the collisions. To collide, at least one of the geometries must be associated with a body. This is necessary, as two geometries without bodies would have no way of showing the result of the collision, since they cannot have forces directly exerted on them. Second, we do not want to collide two bodies that are attached to each other. Otherwise our Joints would be doing a lot of extra work to try and hold things together.

This function does not necessarily mean that two objects are colliding. It simply means that two objects might collide. Calling the Collide(…) function provides a special type of geometry that describes the collision. It is possible that this geometry may not exist – in the case where there is no collision, so before proceeding, we check that the collision actually happened.

For each collision point, we create a special Contact Joint. Remember that joints apply forces on the bodies they are connected to in order to maintain (or reach) some constraint. In this case, the forces are the result of a collision, so by creating the Contact Joint, we are causing the collision that was *detected* to actually happen. It we did not create the Contact Joint, there would be no result of the collision – essentially, no collision would happen.

Synching it back up

ODE bodies have positions as well as orientations which can be accessed at any time. Once we have updated the physics engine by calling World.Step(…), the bodies may have moved. Remember these bodies are associated with some representation in our own program, so now we must synchronize our representation with they physics engine. This can be encapsulated easily enough by maintaining a reference to the ODE body in the

object we use to represent it. That is, if we have a box in our program that is drawn as a six sided cube of side length one, centered at the origin – and we have a and ODE body with an associated ODE box geometry in our physics world – we can access the ODE body's position and rotation, and set our box's Transformation accordingly (translation and rotation).

Conclusion

The use of a physics engine is a great way to make your application much more interesting. There is a lot of complicated math and optimization going on that you really don't want to try and reproduce on your own, and the building blocks are provided for just about anything you might want to do. Integrating a physics engine is not a huge task, and allows you to spend more time on other aspects of your program.

There are a few free physics engines out there, each with their own benefits. Newton attempts to address some of the endless tweaking (so many options, so little time) that you can do in ODE, and uses a slightly solver that is touted as very high precision relative to processing time. Tokamak is the stacking engine. Stacking is a slightly difficult application, as there is a lot of collision happening when you have, for instance, a dozen stacked boxes – let alone a hundred. Novodex may be an interesting one to watch in the future. The company that makes is has recently announced a hardware PPU card. Much like the specialized GPUs that have exploded in the past 10 years (on the consumer level), the Physics Processing Unit – says AGEIA™ Technologies – will take games and consumer level simulation to the next level.

[1] Newton: http://www.physicsengine.com/
[2] Tokamak: http://www.tokamakphysics.com/
[3] Novodex: http://www.ageia.com/novodex.html