2D Physics engine

Embedded Systems Project, EDA 385
Institution of Computer Science

Fredrik Bondza, dt06fb2
Simon Lindgren, dt05sl0
Peyman Pouyan, sx07pp6

18 Oktober 2009

Abstract

This report describes the development of a two-dimensional physics engine as an embedded system on a Digilen Nexys2 FPGA board. The details of both the hardware and software design is presented. Encountered problems and solutions are listed. Finally the lessons and conclusions which this project resulted in is presented.

Contents

1 Introduction

The goal of this project was to implement a basic physics engine which would render all objects onto a screen running on a Nexys2 FPGA board. The engine should simulate gravitational acceleration and collision detection for all of the objects in the world. The user should be able to add objects, via a keyboard, to the world so that multiple objects could be simulated at the same time.

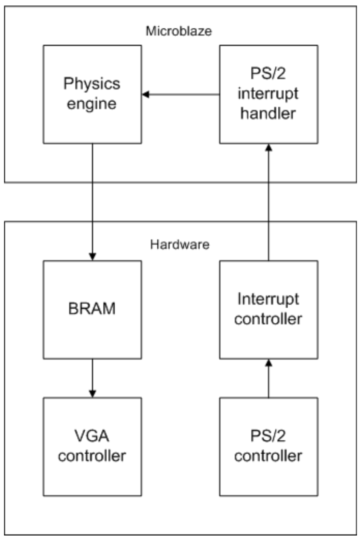

Figure 1: Basic architecture

The basic architecture of the system is shown in figure 1. The hardware consists of a VGA controller with a built-in BRAM, an XPS PS/2 controller, an XPS interrupt controller and a MicroBlaze softcore processor. In software there are two important parts, the physics engine and an interrupt routine.

1.1 Difference compared to proposal

The initial project idea was to be able to simulate objects of different shapes in the physics engine. It was intendet to be able simulate squares and circles. This would require a general way of describing objects so that the collision detection algorthim and drawing algorithm could easily handle both shapes. Because this would make alot of functions much more complicated the engine can not simulate circular objects, instead the engine only handles square objects.

Also there is no collision detection algorithm implemented in hardware. This is mostly because the software collision detection algorithm was very fast and thus it would not have a noticeable affect on the performance.

2 Physics engine

Physics engines are used in video games to simulate realistic behaviour for all objects inside the game. For example to check if a shot hit the target or to calculate how a car will handle when turning into a corner.

The basic physics engine was first implemented on a standard computer and then ported to the FPGA board. This made the development of the engine somewhat simpler since it makes debugging easier. The approach was to first implement physics for a single particle and then extend the engine to handle objects of different sizes.

To make testing of the simulations easier each part of the simulations were added step by step, i.e. first updating position, velocity, acceleration and force of each object. Then a simple collision detection to check for collisions with the world and then object to object collision was implemented.

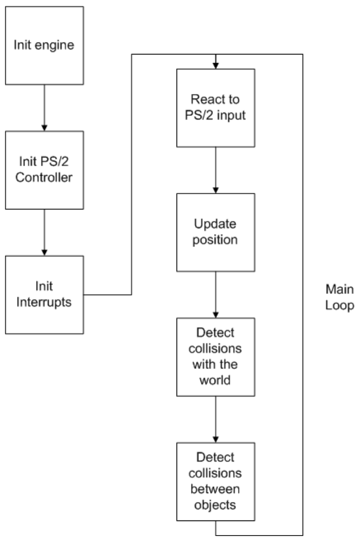

Figure 2: Physics engine dataflow

Figure 2 shows the physics engines different tasks. When the program is started the engine, PS/2 controller and the interrupt routine used to handle PS/2 interrupts is initiated. This means setting up global variables, pointers, vectors and registering interrupt routines which is needed for the program to work as intended.

2.1 World

The world which is visible to the user is represented by the edges of the drawing area on the screen. In the physics engine the size of the world is changed, compared to the screen, to achieve greater precision when performing calculations on objects.

2.2 Objects

The engine can simulate multiple sqaures existing in the world at the same time. Each object has four vectors representing position, velocity, acceleration and force. In order to be able to perform collision detection each of the objects also have the minimum and maximim values of the space it occupies on each axle.

2.3 Gravity

All of the objects in the world is affected by the gravitational acceleration. This is always active and applied everytime an object is updated. Just like in the real world this will make all of the objects fall to the ground, which would be represented by the bottom of the drawing area on the screen.

2.4 Collision detection

There are two basic types of collision detection, object to world collision and object to object collision.

Collision between an object and the world is quite simple since the world is non-moving and in this case it is considered to have inifinite mass. This means that when a collision occurs only the colliding object will be affected. So the colliding object will simply bounce of the ground, like a for example a tennis ball. It will also loose some of its kinetic energy with each collision.

Performing collision detection between two objects is much more complicated than collision between an object and the world because both of the objects can be moving and they can have different masses. This means that the two object should have a different amount of force applied to them when the collision occurs.

A problem which might occur when trying to detect collision between objects is that if the objects have very high velocities they may pass eachother so that they never overlap between two updates, even though they would have collided in the real world. Although in this case the size of the world is limited and relatively small, and the engine has relatively high precision, thus preventing this problem.

3 PS/2 Controller

A PS/2 controller was needed in order to be able to control the physics engine. For example add objects, set object size and remove all objects from the world.

At first a PS/2 custom IP was created to handle input from the keyboard. The architecture of this custom PS/2 controller is shown in figure 3.

This architecture worked on the falling edge of the PS/2 clock then the shift register will store the data. The Finite State Machine(FSM) will look for 11

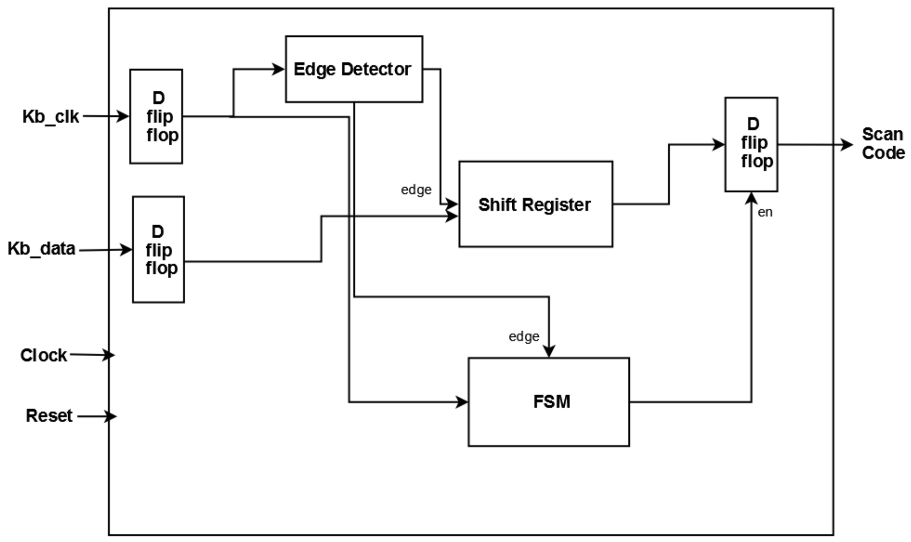

Figure 3: PS/2 Controller design

bits of PS/2 data and when those are found the last flip-flop will output the scan code.

This design could receive input from the PS/2 keyboard as intended. However it would not work with the MicroBlaze processor and an interrupt controller to generate interrupts when a key was pressed.

Therefore this design was abandoned and instead the XPS PS/2 controller combined with an XPS interrupt controller was used from the Xilinx EDK library.

Since the used PS/2 component is part of the standard XPS library the number of slices it occupies is not of any importance for this report thus it is not mentioned.

4 VGA Controller

The VGA controller consists of three major parts, those are the PLB interface and control logic, the BRAM and the VGA controller itself. A bitmap is stored in the BRAM which represents the picture to draw. The processor can write directly to this BRAM to change the image on the screen. The VGA controller reads one word from the BRAM once every 64th clock cycle, with a 50 Mhz clock, to draw onto the screen. Every bit in the memory is directly mapped to a pixel on the screen.

Each of the parts shown in figure 4 is discussed in detail in the following subsections.

The device occupancy for the VGA controller according to the Xilinx EDK was 2% of the total amount of slices.

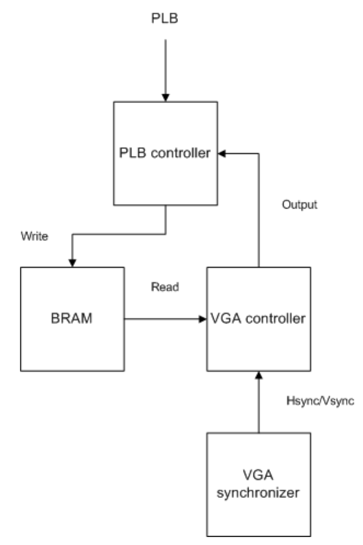

Figure 4: VGA Controller design

4.1 PLB

The PLB is created using the create peripheral function in Xilinx Platform Studio. Then the PLB logic was slightly modified to read from the PLB and write to the VGA controller BRAM. It is only possible to read complete words from the bus, i.e. 32 bits, instead of reading and writing bytes. To make addressing the VGA controller easier from software the addresses sent to the PLB controller is right shifted twice which means that the software can address the VGA controller just like a normal vector.

The downside of this is that when addressing the VGA controllers BRAM it will require four times the amount of addresses when writing to the VGA controller. But the result this is that it will be easier and more obvious how to address the VGA controller from software.

4.2 BRAM

The BRAM component of the VGA controller is configured to only support writes from the PLB and the reads from the VGA controller. This means that there only exists one-way communiction from the MicroBlaze to the VGA controller. Two-way communication is unnecessary because the software does not need to know which pixels that should be shown on the screen.

The size of the VGA controller BRAM is 12.5 KB. The whole picture which should be displayed onto the screen is stored in the BRAM as a bitmap, this means that one bit in the memory directly represents a pixel on the screen. The BRAM address signal is 12 bits wide, this means that it is possible to address up to 16 KB of memory. However there is no functionality to check if this occurs and thus addressing more than 12.5 KB may result in strange behaviour of the whole program.

4.3 Implementation of the VGA Controller

There are two major parts of the VGA controller each of them controls one task.

First there is the synchronizer which generates the vertical and horizontal synchronization signals which is needed for the screen to be able to display the signals. It also generates two signals to keep track of the pixel position on the screen. SIGNALS???

The second part is the controller itself which handles the signals from the synchronizer component to decide which color should be painted for each pixel. The picture on the screen consists of three parts, a background, a border and the drawing area. The background which is just a grid that is drawn where the border and drawing area is not located. Then the border surrounds the drawing area and the drawing area is where the data written from the physics engine will appear.

5 MicroBlaze

The MicroBlaze used in this project had the Floating Point Unit(FPU) enabled in order to be able to do fast floating point calculations. Therefore it is relevant to mention the device occupancy for the MicroBlaze processor including the FPU since that will differ from the normal processor size.

The MicroBlaze processor including the FPU occupied 12% of all avalable slices.

6 Problems and Solutions

6.1 Floating point operations

When developing the physics engine the initial approach was to use floating point numbers to represent the position, velocity, acceleration and force with high precision. The problem this presented was that the calculations could not be done fast enough beacuse of the more complicated nature of the floating point number representation. Even though the hardware Floating Point Unit(FPU) was used with mircoblaze the calculations could not be performed fast enough.

The solution to this problem was to try to eliminate as many of the floating point calculations as possible, since the processor can compute integer operations many times faster than floating point operations. This approach presents another problem which is that the precision is lost when using integers instead of floating point numbers.

To compensate for this the position and size of all objects was multiplied with a constant, which would represent a single pixel. This means that moving an object by the amount of the constant in either axle would make the object move one pixel on the screen. Much like a fixed point representation of a number except that this needs no conversion. The on-screen pixel position is available with just a simple shift operation.

6.2 Limited program memory

The size of the program is limited by the size of instruction memory available on the board. In this case the program was too large to fit into the memory when printout functions was used. This also resulted in debugging issues since no information about objects in the physics engine could be printed.

One of the solutions to this problem is to set the compiler to optimize the size of the program. This means that the size of program might decrease but the performance will also decrease. Another solution is to remove any printout functions since string formatting options and print functions require quite alot of instruction memory. Avoiding unnecessary instructions and assignments is also a good idea, in other words the user optimizes the code himself.

6.3 Synchronize reads from BRAM to VGA controller

Since the VGA controller uses a 32 bit buffer, this means that after a 32 clock cycle period the buffer has to be updated.

The hardware runs at a frequency of 50 Mhz and the pixel position is updated at a rate of 25 Mhz. This combined with the one clock cycle delay of the BRAM created a problem. If a read is performed when there is only one pixel left to draw in the buffer then the buffer will be updated and the pixel will get an incorrect value. If the read was performed on the first pixel in the buffer then the data would arrive too late and old values from the buffer would be used before the new data arrived.

This problem was solved by modeling the read operation as a FSM. The FSM has two states, a read state and a idle state. The idle state is used when there is no need to update the buffer. In the read state, data is read from the BRAM and assigned to the buffer signal. That way the new buffer value will be available the next clock cycle which solves this problem.

7 Possible improvements

7.1 Object rotation

One way to improve the physics engine is to add code which can calculate rotation on objects. This would give a more realistic simulation. However making rotational calculations on objects would probably require to represent objects by just using vectors which means that each update iteration would take longer time. When rotations are taken into account the collision detection part of the engine will not work and would have to be changed to a more advanced algorithm.

7.2 Hardware rendering algorithm

Another improvement would be to move the rendering algorithm into hardware, this would speed up the program but the effect would probably not be noticeable to the user. The current algorithm is very fast due to the fact that the only kind of shape to draw is squares, if other shapes would be introduced the drawing algorithm would probably need to be implemented in hardware.

7.3 Improved collision detection

The collision detection algorithm could be improved by changing it to a more general algorithm which can handle objects of different shapes and rotating objects, i.e. objects at an angle compared to the x and y axis. The current collision detection algorithm is dependent upon how the objects is represented and will not work if all of the objects are represented by just using vectors. Since this would be more complex it would probably also have an impact on performance. Therefore a more complex collision detection algorithm would be better to implement in hardware.

8 Lessons and Conclusions

When software is developed it is easier to start out on a normal computer and then port the software. This is because it is easier to debug the program on a normal computer since there is not the same limitations on instruction memory, therefore printouts are always available.

It is also a good idea to get familiar with the tools before starting a new project, which components are available.

When writing VHDL code always simulate everytime something is added or changed. There is usually some small errors or bugs which will be extremely hard to find otherwise.

9 Installation and user manual

To test this project an archive with the all of the project files are needed. Then the archive needs to be unpacked into a project folder.

Then open the project in the Xilinx EDK using the unpacked project files. When the project is open, choose device in the menu and then in the submenu click update bitstream.

When the system is synthesized and compiled, upload the download.bit file from the implementation folder to the board.

When the program is running on the board then the followin keys can be used. Note that the simulation must be paused before any command can be executed.

Key	*Command*
Space	Pause simulation
Enter	Resume simulation
Esc	Remove all objects
P	Set random speed for all objects
C	Create a new object
W	Move object up
S	Move object down
A	Move object left
D	Move object right
+	Increase object size
-	Decrease object size
0-9	Set object speed

The commands move object, set size and speed can only be used when a new object has been created.

10 Contributions

The following table contains the contributions from each of the group members during this project.

Task	*Responsible*
Physics engine	Fredrik Bondza
Port engine to board	Fredrik Bondza
VGA controller	Simon Lindgren
Rendering algorithm	Simon Lindgren
PS/2 controller	Peyman Pouyan
Section 1	Fredrik Bondza
Section 2	Fredrik Bondza
Section 3	Peyman Pouyan
Section 4	Simon Lindgren
Section 5	Fredrik Bondza
Section 6	Simon Lindgren
Section 7	Fredrik Bondza
Section 8	Fredrik Bondza
Section 9	Simon Lindgren