School of Electrical Engineering

University of Belgrade

GPU Engine

Computer Systems for VLSI course project

Nikola Pejić
0041/2014

Lazar Stojković
0194/2014

Belgrade, June 2018.

Contents

Overview

The GPU Engine represents a framework which contains basic building blocks needed to create games in an HDL language. The implementation of the GPU Engine itself is written in Verilog HDL and was compiled by *Quartus II 64-Bit Version 13.0.1* and tested on an *Altera Cyclone III* FPGA board.

The GPU Engine framework consists of various elements which could be used on their own if one so wishes, but are strategically placed in the current implementation so that the user need only to "tamper" (insert his implement into) a single module (the main module) and abide by its interface in order to develop a simple game, making the usage of the framework intuitive for someone accustomed to writing programs in standard high-level programming languages (such as C/C++, Java, C#, etc.). These various elements are described in the second section.

The GPU Engine code is equipped with three different implementations of games/animations which represent a proof-of-concept for the framework. These implementations are described in the third section.

Lastly, the fourth section contains a discussion on further improvements to the framework that the authors believe can be made, as well as some ideas for further development of the engine.

Composition

The GPU Engine has two types of components:

- General components that are constant across any game
- Game specific components, which are used and combined to create different games

Of course, apart from these components, there are also user-specific ones, that govern the logic of a particular game. However, the components that govern the game logic are often too specific for a single game, and as such were not developed in this engine and are not considered in the further text.

General components

The composition of the constant part of the GPU engine has a structure described in the following subsections.

DEO_TOP

The top entity is the DEO_TOP module used in the Computer Systems for VLSI course, in which the VGA_Output_Controller module is instantiated.

VGA_output_controller

The VGA_Output_Controller module contains the standard elements needed to start developing a game with GPU Engine: the VGA_signal_generator, the Keyboard module with associated Key modules and the main module. The first three kinds of modules are needed by virtually any game, so the main module was introduced with the purpose of encapsulating all the game/user specific implementation.

VGA_signal_generator

The VGA_signal_generator module creates the signals needed to correctly print content on a display. It generates x and y coordinates of the next pixel to be displayed and are used by all the graphical components in creating shapes and colors of the objects.

Keyboard

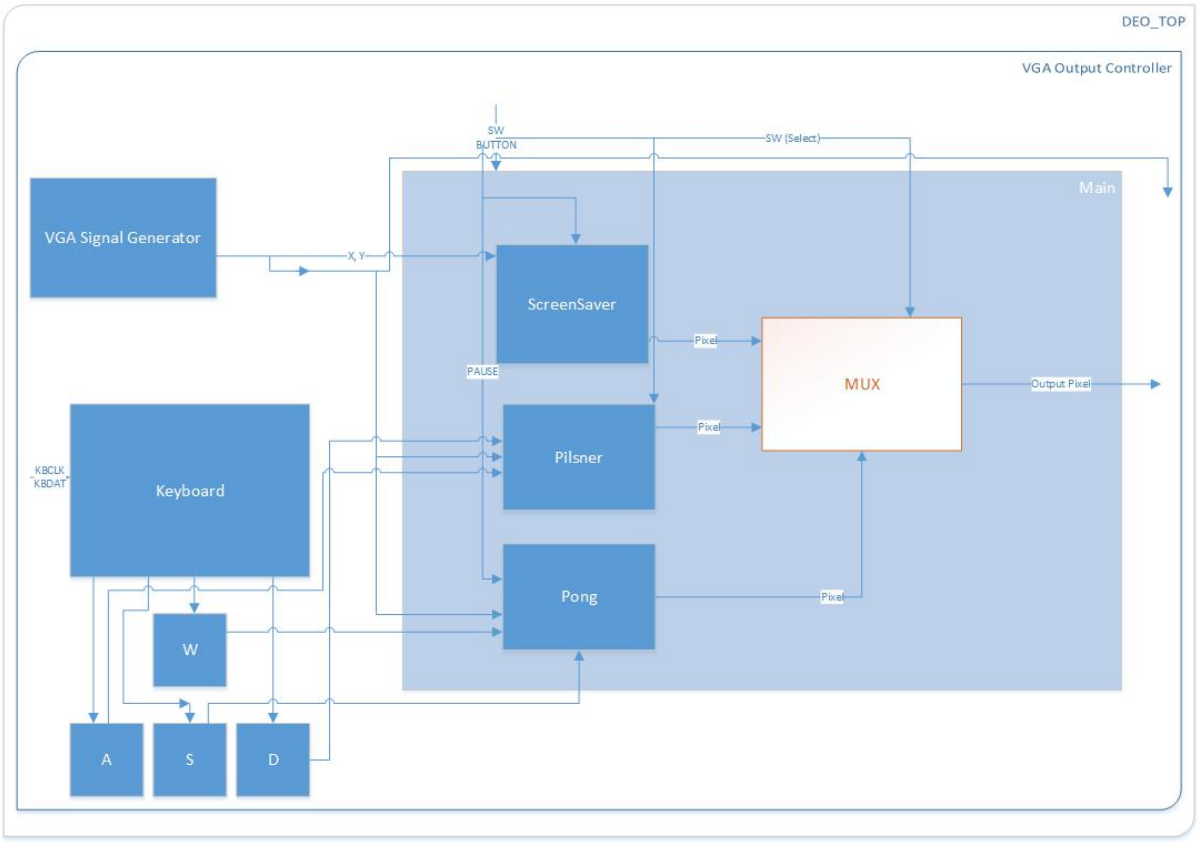
The Keyboard module communicates with a connected keyboard through the PS2 port. It outputs one-byte codes that are used by the Key modules to register if a specific key on the keyboard is pressed or released.

Key

The Key modules are simple state machines that listen to the output of the Keyboard module and change their state from OFF to ON and back to OFF if a specific code (or sequence of codes) is registered that indicates the key being pressed or released. The output of these modules is a single Boolean value that represents the current state of the key (ON/OFF, Pressed/Not Pressed).

Main

The main module is the module which contains the user/game specific implementation. It uses the output of the other modules in the VGA_output_controller to generate a pixel for every x and y coordinate of the screen. At a high level, it basically represents a function f(x,y,t), where x and y are coordinates on the screen, and t is time.



High-level overview of the GPU Engine design

Game specific components

The game specific components represent modules that are general enough to be developed and put into the engine, and that can be combined to create specific content of a game.

The basic modules that have been developed mimic the object-oriented nature of higher-level languages (like C++, Java, C#, etc.), as they are used to create an individual object on the screen. That is, the object is created by instantiating the modules and initializing them and can then be left to act on its own (it "has a life of its own", so to speak) and will need only minor alterations in order to abide by more specific rules than the ones envisioned in the engine. An example of how an animation can be made by only initializing the object and not doing any modifications on them afterwards is the POC *Screen Saver* module.

The basic modules are described more specifically in the following subsections.

Physics

The physics module controls the physics of an object. It has a position (x and y coordinates) and speed (x and y components). With a specific period (predefined to *GLOBAL_PHYSICS_PERIOD = 10 ms*) the physics module updates the current position of the object by adding the speed factor to the respective position coordinates. The current implementation of the physics module is purely kinetic, as there is no acceleration component.

The physics engine can also have predefined minimum and maximum values for the position coordinates. When the object goes out of bounds, the module has a couple of modes of operating that were thought to be useful during the designing of the module, which are listed below:

- No Action – when the object goes out of bounds, it is snapped to the nearest valid value (which is the bound that it most recently crossed). In this mode, it appears as if the object that was moving hits a wall and stops (as long as the speed of the object remain the same).
- Reset – when the object goes out of bound on one of the axes, the value of that axis is reset to the initial value (defined on instantiating the object).
- Full Reset - when the object goes out of bound on one of the axes, the value of both axes is reset to their initial values (defined on instantiating the object).
- Bounce – when the object goes out of bound on one of the axes, the position of the object does not change, but the speed component of the object that lead to the out of bounds event is inverted, basically bouncing the object from the bounds.

Renderer

The renderer module is used to define the shape of the object. For the current x and y coordinates and the coordinates of the top left corner px and py (obtained from the physics module) the renderer module outputs a single value that indicates whether the current pixel should be printed. Basically, the module is a function f(x, y, px, py, t) which, for the position of the object (px and py) divides the screen into two domains, one that represents the object, and the other that doesn't represent the object.

The renderer has some predefined shapes of objects that it can render, which are presented in the following list:

- Square
- Circle
- Triangle
- Rectangle
- Text

All the shapes are constructed using an integer as an indicator of the size, where the rectangle has a second size value (in order to have sides of different sizes), as well as the text shape, which uses the second size value as an input channel for the codes of the values that will be displayed.

The renderer also has the ability to shrink or enlarge the object, as an additional component (the speed of the resizing) can be defined which, much like the physics module, changes the value of the sides over time.

Shader

The shader is a module that is used to provide the color for the object. Currently, it only outputs a single color for the whole object, but the goal was to have a color gradient which would go from one shade to another as the x and/or y coordinates increase.

The shader module outputs the level of the object, which is used by the pixel_join module to create depth – if two objects should be intersect, the level defines which of them will be shown at the intersection.

Pixel Join

As a standard object consists of a physics, renderer and shader module, and every object outputs a value for the pixel at any given time, a mechanism is needed to somehow aggregate the pixels. The module that is used for this purpose is the pixel_join module. This module takes the output signals from every object and decided which pixel is going to be shown. If no pixel is active, then a default

pixel (the background pixel defined when instantiating the module) is set as the pixel that is going to be shown.

Util modules

There are a few util modules that were used inside other modules. These modules are listed in the following subsections.

Bound checker

The bound checker module is used to check if a given input value is within some predefined bounds. If it is not, then it indicates this as an output, along with the snap value, which represent the value with the smallest offset from the input value which is in a valid state (is within bounds).

Square wave generator

This module, as its name suggests, generates a square wave of a predefines period (defined when instantiating the module). Apart from the actual wave, it also output a signal indicating if a rising edge of the wave was detected.

Collision counter

The collision counter module is used for counting the number of collisions of object. Basically, it has a bit vector as input, where the bits indicate if an object is active (outputs a pixel). It then counts the number of objects that are active and return this value.

This module was used in making the intersections of objects have different shades in the *Screen Saver* animation.

Proof of Concept

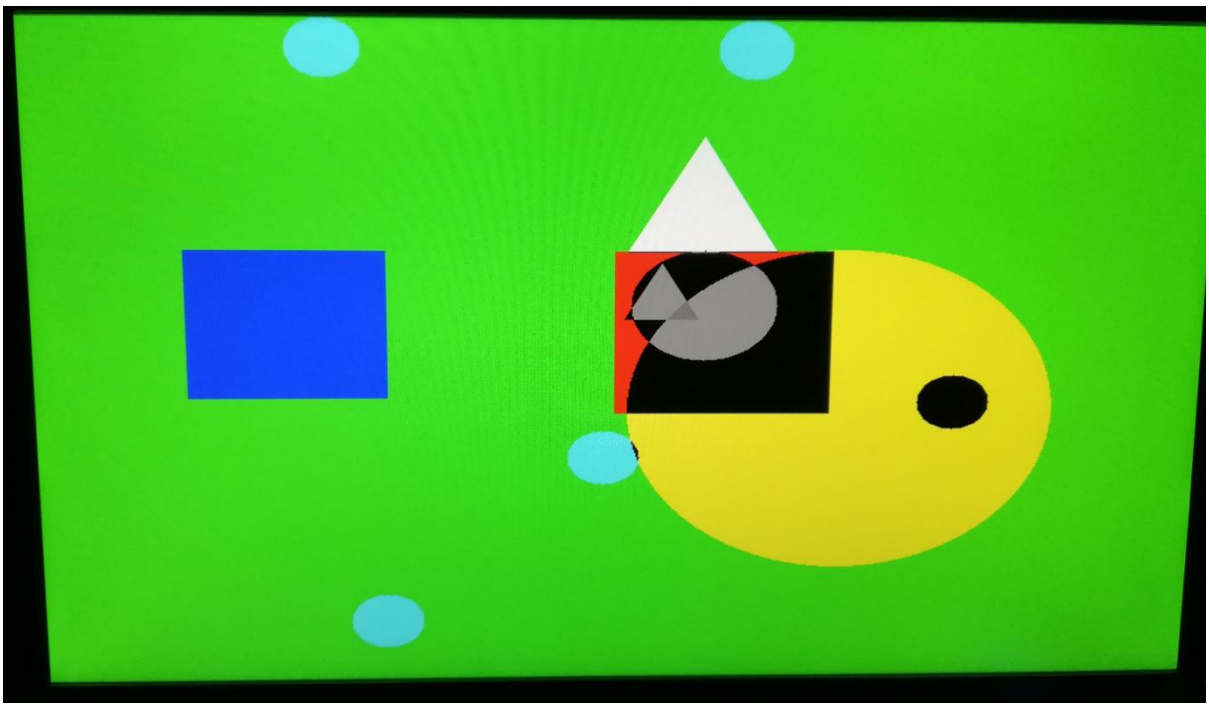
As a proof of concept for the GPU Engine, two games and an animation were made. They showcase the different features of the engine and are described in the next sections.

Screen Saver

The Screen Saver is an animation which showcases the various modes the objects (their modules) can have.

It consists of various object (squares, circles and rectangles) with different colors flying over the screen at different speeds. Some of them have the resize feature of the renderer turned on, so they shrink or grow at some pace. Some of them have the reset and full reset modes turned on, so they follow some pattern in their movement. Some of them have the bounce mode turned on, and bounce of the sides of the screen.

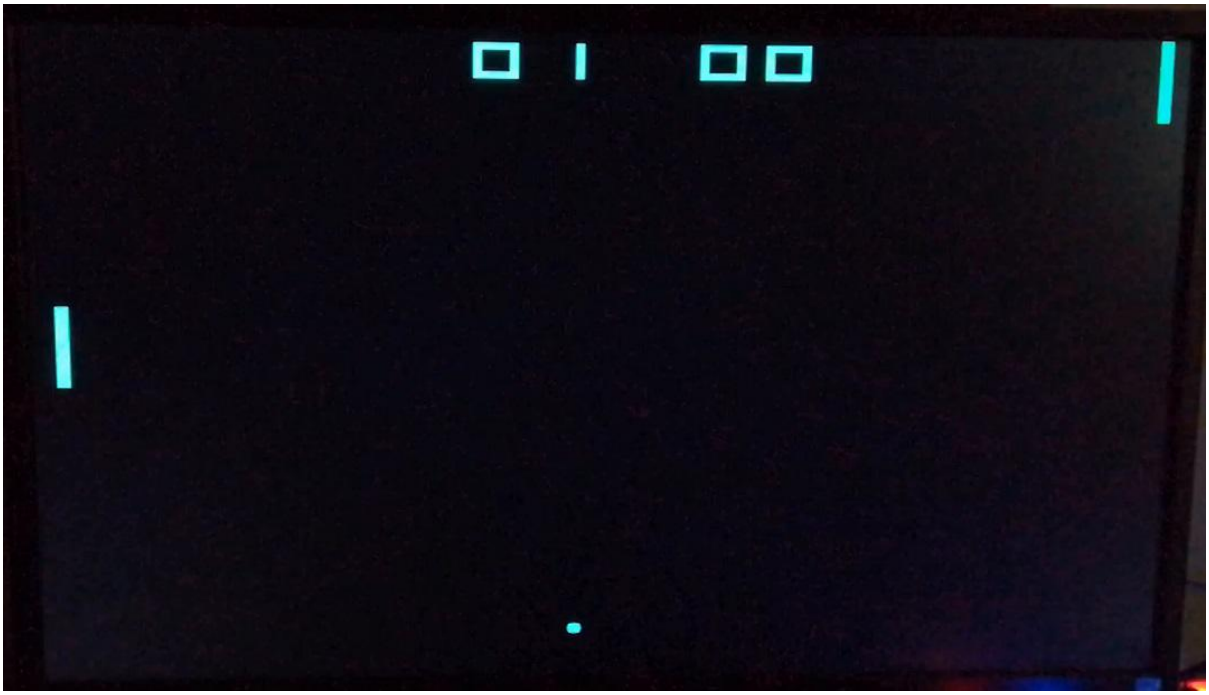
The interesting feature that is showcased in the Screen Saver POC is the collision counter module. Basically, if a collision is detected, the number of object that are intersecting is counted and a color from a palette array is picked for the color of the intersection.



Screen Saver - showcase of intersecting objects

Pong

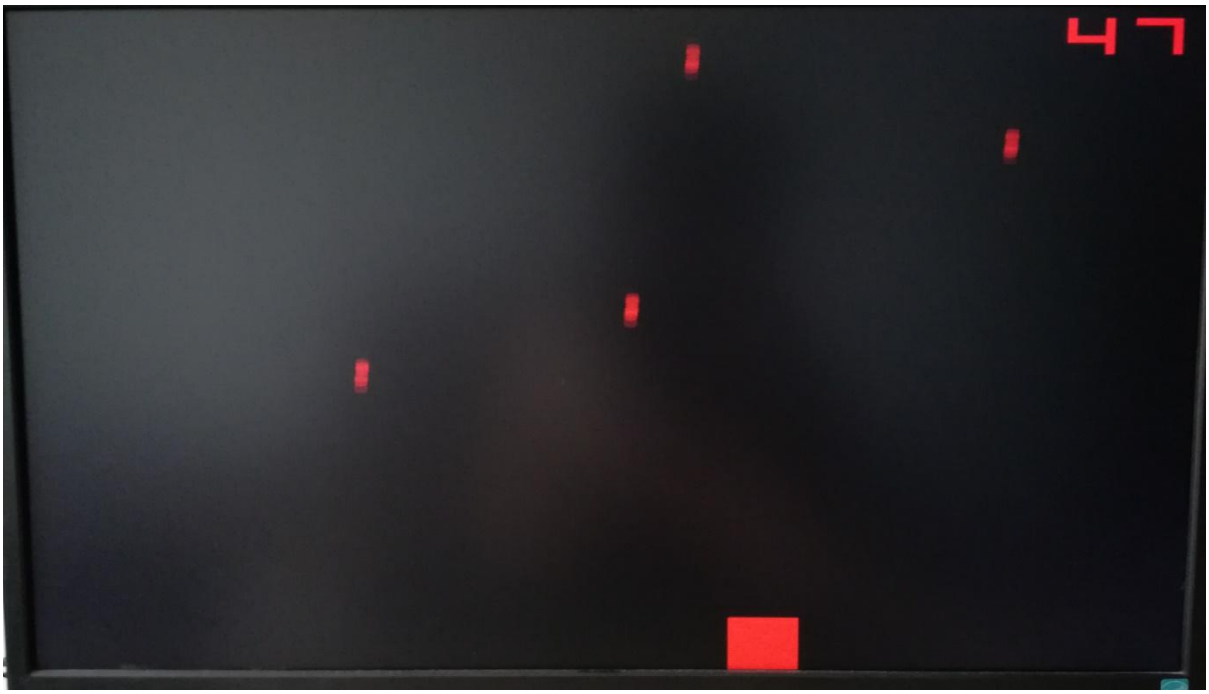
Pong is a simple two player game. The players control pads which are placed on opposite sides of the screen and which can move up or down by pressing appropriate keys. The players try to bounce the ball which is moving at some speed. The goal of the game is bounce the ball the most number of times, as when a player fails to bounce it (when he misses the ball and it goes past the pad) the other player gains a point. A scoreboard at the top of the screen shows the current score.



Pong – gameplay

Pilsner

Pilsner is a single player game where the player controls a square located at the bottom of the screen that can move left and right. There are small circles falling from the top of the screen. The goal of the game is to collect as many circles as he can, thus gaining the most points.



Pilsner – gameplay

Future work

In this section some ideas on what additional features could be developed for the GPU Engine, and what could be some future courses for the engine.

Some features that could be developed are listed below:

- The physics update is done regardless of the fact if the current frame is finished printing or not. Because of this, the objects may at times seem like they are broken, having one part shown in the previous position, and the rest of the object being in the new position. This could be fixed by waiting for the frame to finish being displayed and then updating every physics module.
- The current implementation only supports PS2 keyboard input. It would be a good feature to support the PS2 mouse as well.
- The resolution of the screen is such that it makes the objects wider that they should – for example, a circle appears as an ellipse, and a square as a rectangle. This bug should be fixed in order to have a better quality of displayed objects.
- The current implementation of the shader module outputs a pixel with constant color. It was envisioned that the shader would have the ability to have gradient colors, or have some small color pattern which it would then use to fill the whole object.

An idea for future work on this project is to make it more GPU-like – that is, to have a unit that would read instructions from some input source (be it a memory or a bus) and change the states of the object and initialize them according to those instructions. It would then mimic the function of the GPU and could be programmed simply by altering the instruction source, without the user having to know any of the HDL languages.