

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

MASTER'S THESIS

Implementation and simulation of two
pass forward texture mapping on a
SIMD coprocessor architecture

Martijn van Sinten

Supervisors: dr. M.R.V. Chaudron
dr. ir. A.J.F. Kok
dr. A.A.M. Kuijk
prof.dr. F.J. Peters

Eindhoven, August 2002

Preface

I really enjoyed my time at Philips, and I would like to thank everyone at the graphics cluster for that. I would also like to thank my supervisors from the university, Michel Chaudron and Arjan Kok.

Contents

1	Introduction	3
1.1	<i>Rendering</i>	3
1.2	<i>Texture mapping</i>	3
1.2.1	Forward and inverse mapping	3
1.2.2	Two pass forward texture mapping	4
1.3	<i>Image Blender</i>	6
1.3.1	Components	7
1.3.2	Execution order	8
1.4	<i>Problem description</i>	9
2	Simulating the Image Blender	10
2.1	<i>Simulation Framework</i>	10
2.1.1	Component	10
2.1.2	Connector	12
2.1.3	Connection	12
2.1.4	Simulation	13
2.2	<i>Simulation GUI</i>	15
2.3	<i>An example of a simulation</i>	16
3	Implementation of two pass forward texture mapping on the Image Blender	19
3.1	<i>Two pass forward texture mapping on the Image Blender</i>	19
3.1.1	Distribution of the texel processing	19
3.1.2	Resampling equation	21
3.1.3	Processing the texels	22
3.1.4	Calculation of the texel borders	24
3.1.5	Implementing the texel processing	25
3.1.6	Using the PE's to process a block of texels	28
3.1.7	Number of PE's	29
3.2	<i>Simulating two pass forward texture mapping on the Image Blender</i>	29
3.2.1	Image Memory	29
3.2.2	Processing Element	29
3.2.3	Lookup Table	31
3.2.4	System Controller	31
3.2.5	Sample Mapper	31
3.2.6	Connections	32
4	Performance comparison	33
5	Further work	33
6	Conclusion	34

1 Introduction

The problem addressed in this report is the implementation of a two pass forward texture mapping algorithm on a SIMD (Single Instruction/Multiple Data) coprocessor architecture called the Image Blender. A simulation of this implementation is made to work out the details, find problems and test the performance of possible different implementations.

1.1 Rendering

Rendering is the conversion of an three-dimensional object-based description of a scene into a two-dimensional image for display. This description contains information about the scene like the positions and orientations of the virtual camera, the three-dimensional objects and the light sources. These objects are instances of models. Models are defined in their own coordinate system called modelspace. Every instance of a model is transformed such that all objects reside in a common coordinate system called worldspace. Because a scene can be viewed from different locations and at different angles the scene is transformed into yet another coordinate system called eyespace. This transformation depends on the position and orientation of the viewer. After this the scene is projected onto the viewing plane: all 3 dimensional coordinates are mapped into 2 dimensional coordinates with a depth value. This 2 dimensional description is then used to assign the correct color values to the pixels (picture elements) of the output space, most likely a screen (then often called screen space).

1.2 Texture mapping

The level of realism of the rendered scene can efficiently be increased, without increasing the geometrical complexity of the scene, by mapping an image onto the surface of an object. The process of mapping an image onto a geometry is called texture mapping. The image is called a texture and its pixels are called texels. Texture mapping is not limited to just image based mapping, but can also be used for mapping bumpmaps, lightmaps, functionmaps, etc. onto an object. In this document texture mapping is restricted to image based texture mapping.

1.2.1 Forward and inverse mapping

There are different ways to perform texture mapping: inverse texture mapping and forward texture mapping. The difference between both methods is shown in figure 1.

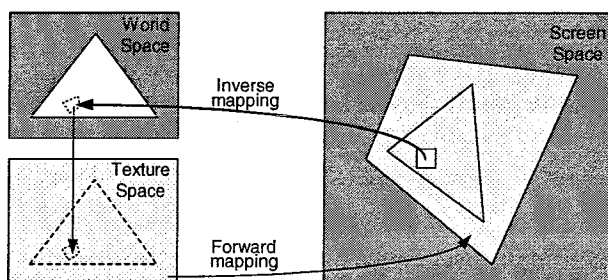


Figure 1: Forward and inverse texture mapping.

Inverse texture mapping is the more commonly used method. For every pixel in the output space the contributing texels are determined by inverse mapping the area of the pixel onto

the texture image. The (weighted) samples of this area are used to determine the resulting contribution for the pixel shading. It proves to be quite difficult to combine these samples in the right way in order to avoid artifacts.

Forward texture mapping transforms a texture image until it conforms to the projection of the image on an arbitrarily oriented surface in 3-space. In other words, for every texel it is determined to which pixels this texel contributes, instead of the other way around which is the case with inverse texture mapping.

From here on the surface on which the image is projected is assumed to be a plane. The main advantage of forward texture mapping is that it is potentially much more efficient and therefore faster than inverse texture mapping. The access to the memory containing the texture images is very regular because texels can be processed sequentially. Every texel is then needed only once, so this has the potential to save memory bandwidth, a common bottleneck in the graphics hardware today. For this reason this method is a subject of studies for hardware implementation at Philips NatLab. Consequently, the focus of this study is on forward texture mapping.

1.2.2 Two pass forward texture mapping

Forward texture mapping can be performed in two separate passes [Catmull-Smith80], utilizing an intermediate image. Here, first all horizontal lines of the image are mapped onto the intermediate image. In the second pass all vertical lines from the intermediate image are mapped onto the final output image. The method of two pass forward texture mapping will now be explained in more detail, beginning with the first pass:

First pass

The input texture image consists of texels with coordinates (u, v) . In the first pass the rows of the input image are mapped onto corresponding rows of the intermediate image. To accomplish this, the (u, v) coordinates are transformed into (x, v) coordinates, where v is left unchanged and x is calculated using the next equation [Catmull-Smith80]:

$$x = \frac{Au + Bv + C}{Gu + Hv + I} \quad (1)$$

where it is assumed that the homogeneous perspective transformation matrix is formed by

$$\begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix} \quad (2)$$

This process is depicted in figure 2. Texels $t_{i,v}$ from the input space are mapped into the intermediate space. This mapping is shown by the open dots in the figure above. These are not equidistantly spaced because of the perspective projection. How these mapped texels contribute to the pixels in the intermediate space is explained next. ($p_{3,v}$ is used in figure 3)

A filter function is used to calculate a weighted average of all texels from the input image that contribute to a pixel in the intermediate image. Because this is a weighted average, the

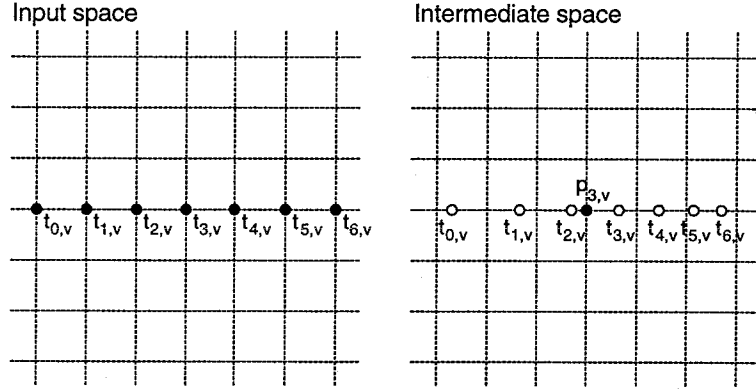


Figure 2: The first pass.

total area enclosed by the filter function is equal to 1. The value of a pixel p can be calculated using [Meinds00a]:

$$C_p = \sum_{t=b}^e (H(M_{t+1} - X_p) - H(M_t - X_p)) C_t \quad (3)$$

In this equation b and e represent respectively the first and the last texel that contribute to pixel p . How much each texel t with value C_t contributes to pixel p depends on the contribution factor, calculated with $H(M_{t+1} - X_p) - H(M_t - X_p)$. $H(x)$ is the primitive of the used filter function, i.e. $H(i) = \int_{-\infty}^i h(x) dx$, and M_t represents the midpoint between texel t and $t - 1$. This value can be approximated by $M_t = \frac{1}{2}(X_{t-1} + X_t)$. X_t and X_p are the positions of respectively texel t and pixel p in the intermediate image coordinate space. The application of this equation is shown in a graphical way in figure 3.

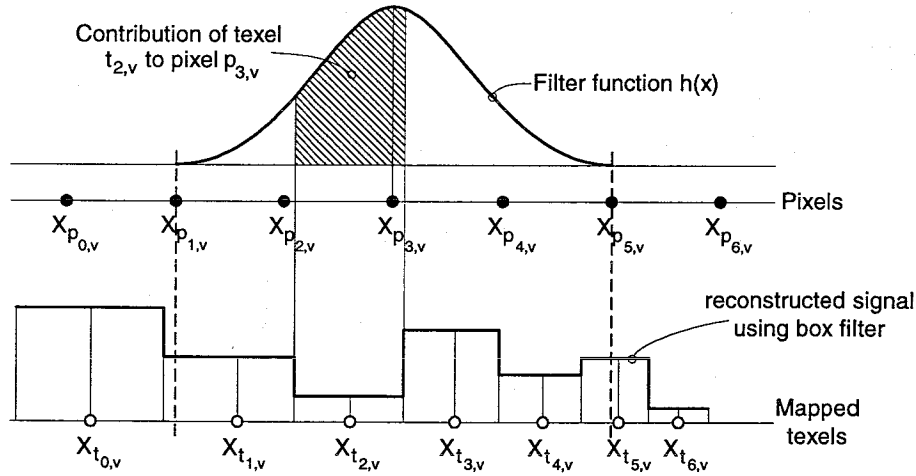


Figure 3: Calculation of the contributions to pixel $p_{3,v}$.

Here $t_{1,v}$ is the first and $t_{5,v}$ the last texel that contribute to pixel $p_{3,v}$. $X_{t_{i,v}}$ is the position of texel $t_{i,v}$ in the intermediate image coordinate space. An example of the first pass is shown in figure 4.

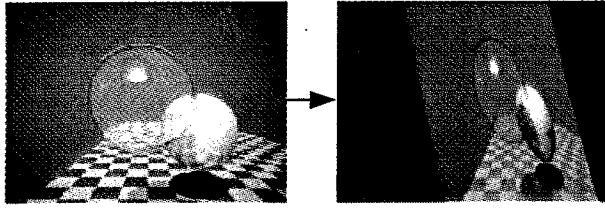


Figure 4: Example of the first pass [Catmull-Smith80].

In the second pass the columns of the intermediate image are mapped onto corresponding columns of the output image. Here the (x, v) coordinates are transformed into (x, y) coordinates, with x left unchanged and y calculated using [Catmull-Smith80]:

$$y = \frac{(Dm + E)v + (Dn + F)}{(Gm + H)v + (Gn + I)} \quad (4)$$

with $m = \frac{B - Hx}{Gx - A}$ and $n = \frac{C - Ix}{Gx - A}$.

The second pass is depicted in figure 5. Filtering is done in the same way as in the first pass. An example of the second pass is shown in figure 6.

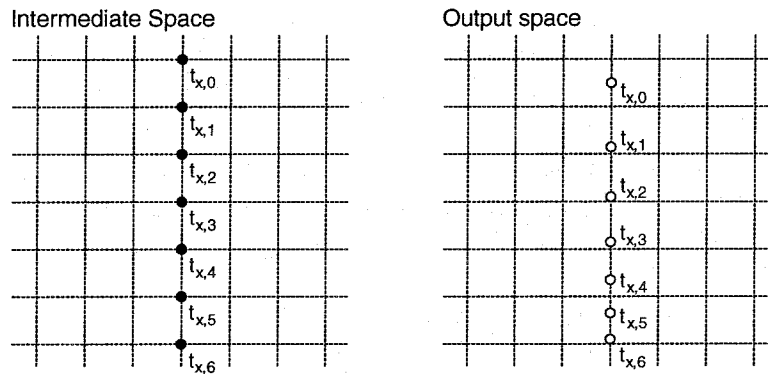


Figure 5: The second pass.

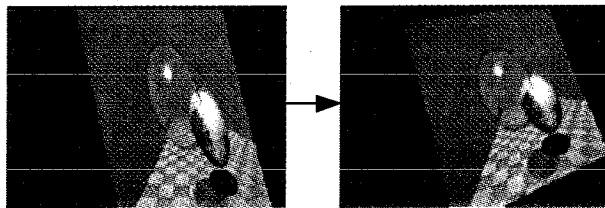


Figure 6: Example of the second pass [Catmull-Smith80].

1.3 Image Blender

The Image Blender [Kuijk00] is a SIMD coprocessor architecture for pixel based image synthesis. It is developed to take advantage of the coherency of adjacent pixels and texels that often occurs in image processing. The most important part is an array of processing elements

(PE's) which can perform relatively simple operations on pixels and texels. These PE's get data from a local memory, the Image Memory. The PE's are connected to the Image Memory via a high-bandwidth connection. This connection is provided by the Sample Mapper. Each PE also has access to a lookup table which it shares with the other PE's in the array. The whole ensemble is controlled by the System Controller. Figure 7 gives an overview of the Image Blender architecture.

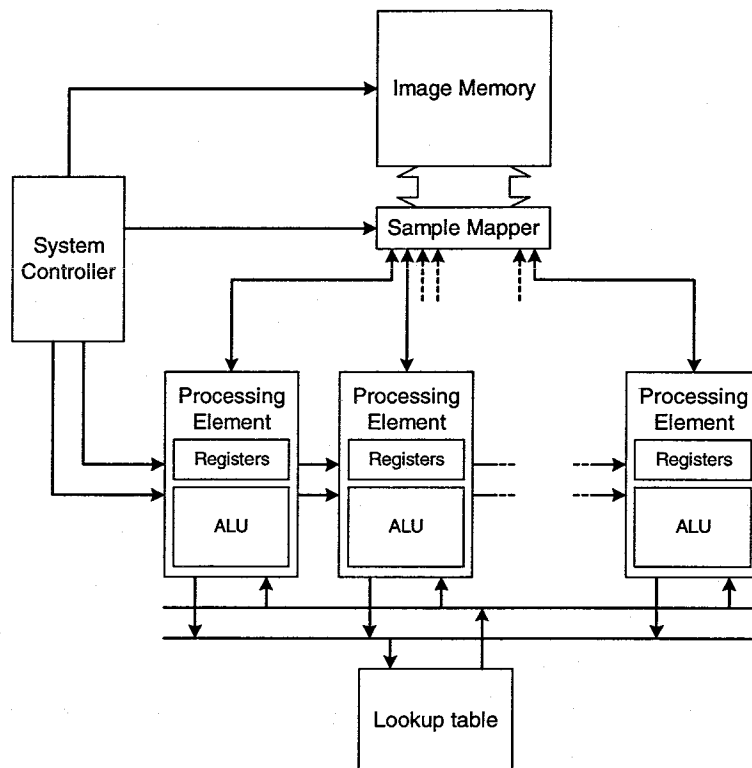


Figure 7: An overview of the Image Blender.

1.3.1 Components

All the separate components of the Image Blender are now discussed in more detail:

System Controller

The tasks of the System Controller are to

- issue the instructions for the PE's,
- determine which subset of the PE's execute the instruction,
- provide data to the PE's accompanying the instruction,
- control the Image Memory.

An alternative route via the Sample Mapper is also available to distribute data from the System Controller to the PE's.

Processing Elements

Processing Elements execute instructions issued by the System Controller. The instructions are basic instructions that take at most one cycle to execute, for instance the addition of two

vectors, or the scaling of a vector. Instructions can be conditional, i.e. the execution depends on the value of a particular register. Data can be received via the Sample Mapper, the Lookup Table and the previous PE. Output data can be sent to the Image Memory, the Lookup Table (i.e. selecting an index) and the next PE.

Image Memory

The Image Memory may hold pixel images, texture images or function values. It consists of 27 bit elements and is organized as a VRAM (Video Random Access Memory). Images are made up of multiple layers to support different kinds of values, i.e. one layer for color (i.e. 3 signed color components of 9 bits), one layer for opacity (alpha value), one layer for depth (z-value), etc.

Sample Mapper

The Sample Mapper provides the link between the PE's and the Image Memory. It provides each PE with an element from an array of data from the Image Memory. This array of data is usually a section of a scanline. The Sample Mapper is also used to put processed data from the PE's back in the Image Memory.

Lookup Table

The Lookup Table provides a lookup facility for e.g. environment maps and function values to replace costly calculations. This is common practice for operations such as exponentiation, normalization of vectors and noise functions.

1.3.2 Execution order

Instructions are issued by the System Controller. All instructions are executed in precisely one cycle. Usually for SIMD systems each instruction is executed on all PE's at the same time. Alternatively one can introduce a time delay so that each PE executes an instruction one cycle after the previous PE. Then each PE passes the current instruction on to the next PE.

Introducing this time delay has the advantage that the lookup table can easily be shared between the PE's: each instruction that needs access to the lookup table is executed on each PE one cycle after the previous PE executed this instruction. So the lookup table is shared automatically as long as the next instruction that needs access to the lookup table is executed at least N cycles later, where N is the number of PE's. If all instructions would be executed at the same time, a simple bus arbitration method involving time delay circuitry would have to be introduced. This implies that the result of the lookup would not be available until N cycles after the lookup instruction.

When execution of the instructions of neighbouring PE's are delayed one cycle, the input data for these instructions needs to be delayed accordingly. After processing, this delay has to be compensated, in order to have the Sample Mapper receive all results from an instruction at the same time. This is usually not a problem as long as the data is not needed within the next N cycles, where N is the number of PE's. The delay can easily be introduced by using delay lines consisting of latches (a digital logic circuit which is used to store one or more bits of data) that store the data for one cycle, as is shown in figure 8.

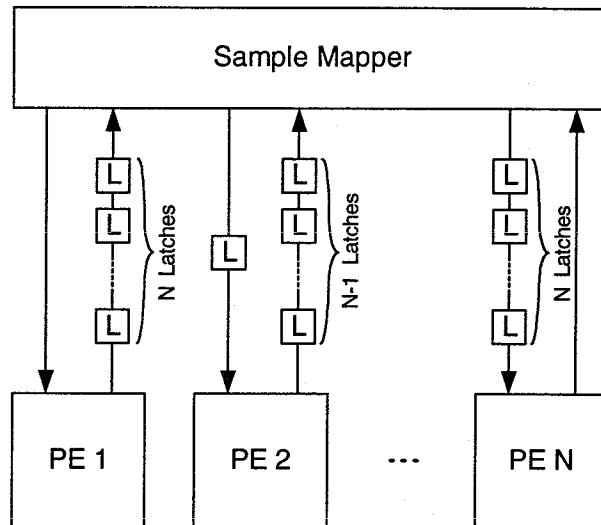


Figure 8: Added delay's between the Sample Mapper and the PE's.

1.4 Problem description

In [Kuijk00] an architecture is described where the Image Blender is combined with a separate unit for perspective texture mapping. This separate unit takes care of the texturing operations and the Image Blender takes care of the non-texturing operations. However, this scheme implies that the ratio between the number of PE's and the number of perspective texture mapping units is fixed, and therefore may lead to suboptimal use of hardware resources.

The possibility to implement two pass forward texture mapping on the Image Blender needs to be examined. This is desirable because then all operations can be done on the Image Blender and optimal use of hardware resources is obtained automatically.

The focus in the implementation of two pass forward texture mapping is limited to minification only, because magnification can be done in the same way by upscaling the texture. This means that the 'width' of a texel, i.e. $M_{t+1} - M_t$, is less than or equal to the space between two subsequent pixels. Since pixels lie on an integer grid this implies: $M_{t+1} - M_t \leq 1$.

A simulation of this implementation needs to be made to work out the details, find problems and test the performance of possible different implementations. The speed of the simulator itself is therefore not very important. It will be implemented in Java.

2 Simulating the Image Blender

The Image Blender is an architecture proposal. Therefore it is possible that certain details will be changed. It is also possible that changes are actually necessary to implement two pass forward texture mapping. So it would be practical if the simulator that is constructed can be easily modified and extended. Therefore a component based simulation framework is chosen: the simulation is built from components which are linked together by means of connectors and connections. This is depicted in figure 9. An advantage of using separate components is that the behavior of each component can be defined independently of the behavior of other components. Within one simulation cycle the components are not influenced by each other.

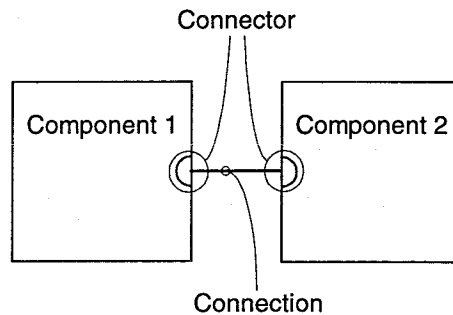


Figure 9: The framework components.

2.1 Simulation Framework

A simulation consists of a collection of components which have connectors and are connected through these connectors via connections. Those building blocks are implemented by deriving new ones from the base classes Component, Connector and Connection. The base classes of the simulation framework are shown in the class diagram of figure 10. These base classes are further described in the next subsections.

2.1.1 Component

Components are the building blocks of a simulation. Each component has one or more connectors to link itself to other components. All components are defined independently. A complete system can then be composed by linking instances of these components together. Such a system can be easily extended or adapted by adding new components or replacing others. A component is derived from the class Component:

```
public abstract class Component
{
    private String name;

    public Component(String name)
    public void processCycle()
    public String toString()
    public String getName()
}
```

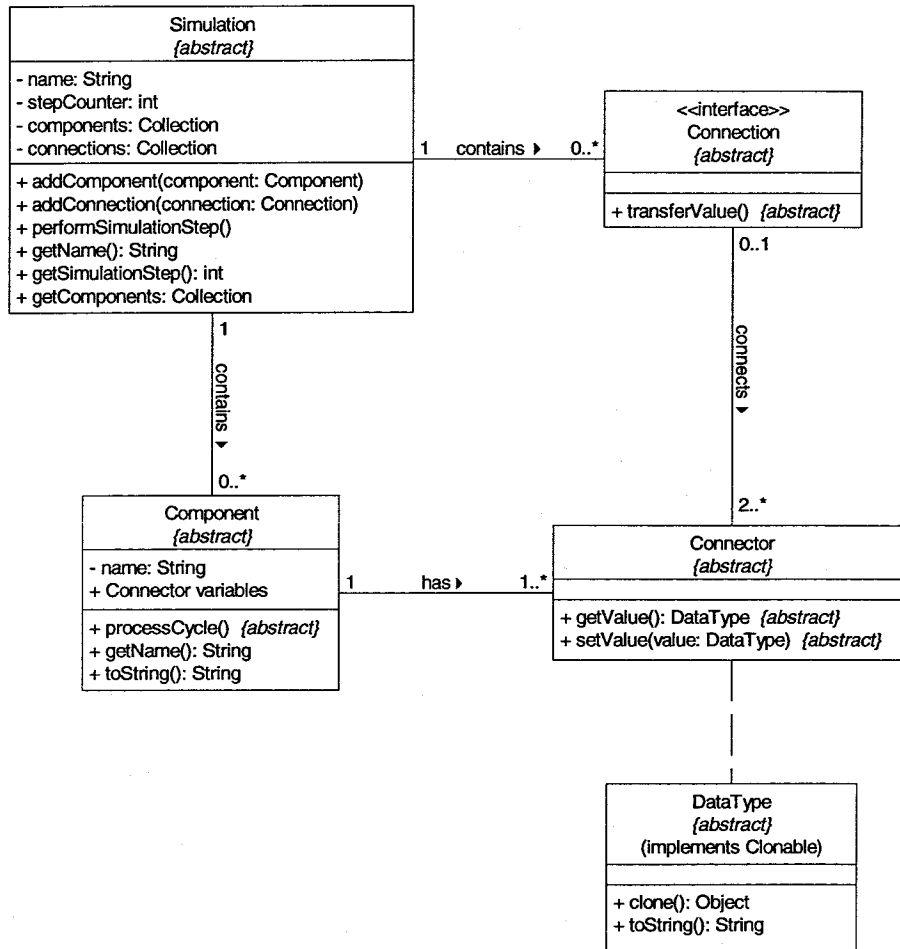


Figure 10: Full classdiagram of the simulation framework.

The `processCycle` method is the most important method: it defines the components behavior, i.e. how it uses the values from its inputs and which values it puts on its outputs. Typically this method consists of 3 steps:

- read the input values from the input connectors,
- perform some operations on these values,
- write output values to the output connectors.

The `toString` method is used to get information from a component, like the current state, in the form of a textstring. The method `getName` returns the value of the `name` variable which contains the name of the instance of this component. This method is used by the interface of the simulator to list the components of a simulation.

The implementation of `processCycle` in the base class `Component` is empty. The other three methods all have simple implementations. The constructor just copies the value of its only parameter `name` to the variable `name`. `toString` returns the textstring "`<undefined representation>`" by default. This method needs to be overridden to return a more useful value, which fits the state of the newly defined component. The implementation of the

getName method only consists of returning the value of the name variable.

The interface of a component to other components is defined by its connectors. These are public variables to which instances of a class derived from Connector are assigned.

2.1.2 Connector

Connectors are used to define the interface of a component to other components in the system. Each connector holds one value that can be read and written. A connector is derived from the class Connector:

```
public abstract class Connector
{
    public abstract DataType getValue();
    public abstract void setValue(DataType value);
}
```

The value held by a connector can be read with the getValue method and set with the setValue method. This value is an instance of a class derived from DataType:

```
public abstract class DataType implements Cloneable
{
    public Object clone()
    public String toString()
}
```

This class is needed to have a common way to transfer values between connectors. New datatypes can be defined by deriving a new class from this base class. If this new class contains any reference variables, the clone method has to be overridden. The new method must then take care of the duplication of these reference variables. The clone method is used to duplicate DataType objects.

The toString method defines how the value is represented by a textstring. Its implementation in the base class just returns the value "<undefined representation>". Therefore it needs to be overridden by derived classes to return a meaningful value.

2.1.3 Connection

Connections are used to link a number of instances of components together to form a complete system. A connection links output connectors to input connectors. Each connection is an instance of a class that implements Connection interface. The specification of this interface is the following:

```
public interface Connection
{
    public void transferValue();
}
```

The `transferValue` method defines how values are transferred from the output connectors to the input connectors. This can be seen as defining the behavior of a connection, similar to implementing the `processCycle` method of a component. In fact, connections are exactly like components, except that the operation performed by a connection is not accompanied by a delay, as opposed to a component. The amount and the type of input and output connectors is defined in the constructor of a class that implements the `Connection` interface.

A basic connection, which can be used to connect two connectors of the same type, is defined by the `BasicConnection` class. This class implements the `Connection` interface. The implementation of the `transferValue` method consists only of copying the value from the output connector of the source component to the input connector of the destination component. The constructor accepts two parameters, the first one defines the source connector, the second defines the destination connector. This defines the direction of the connection.

Besides this simple connection, a wide range of different kinds of connections can be defined. Some examples:

- demultiplexor (a connection between one source connector and multiple destination connectors)
- splitter (for instance, a connection that splits a RGB value held in a connector into its separate monochrome components)
- shift (a connection that performs a bitwise shifting operation)

Schematic representations of these examples are shown in figure 11.

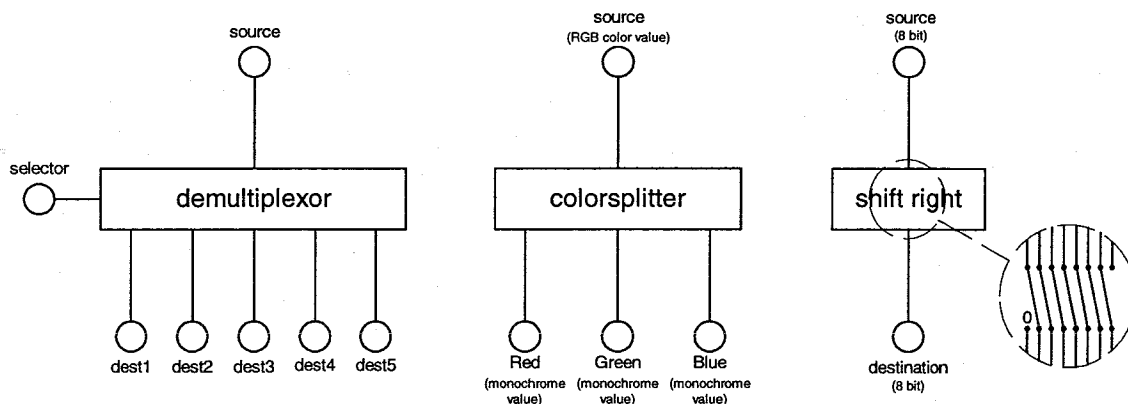


Figure 11: Example of a demultiplexor, a colorsplitter and a shift.

2.1.4 Simulation

A complete simulation is constructed by deriving a new class from `Simulation` and defining the structure to be simulated. First components are instantiated and added to the simulation with the `addComponent` method. Then Connections can be made between those components and added to the simulation with the `addConnection` method.

When everything is added, the simulation can start. By calling the `performSimulationStep` method, one simulation step is performed. In each simulation step first the `processCycle` method of every component is called. After this step all components have performed their task (for that cycle) and transferred the output values to the output connectors. Then the

transferValue method of every connection is called, which transfers the output values to the input connectors of the involved components. This completes the simulation step. This process is depicted in figure 12 as an activity diagram and in figure 13 as a sequence diagram.

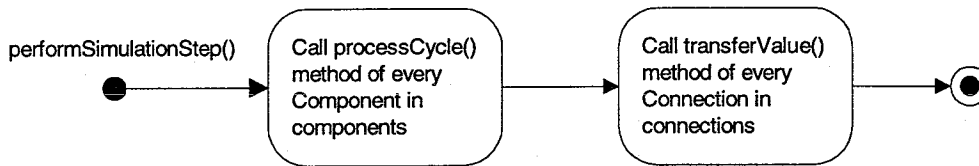


Figure 12: Activity diagram of performSimulationStep method.

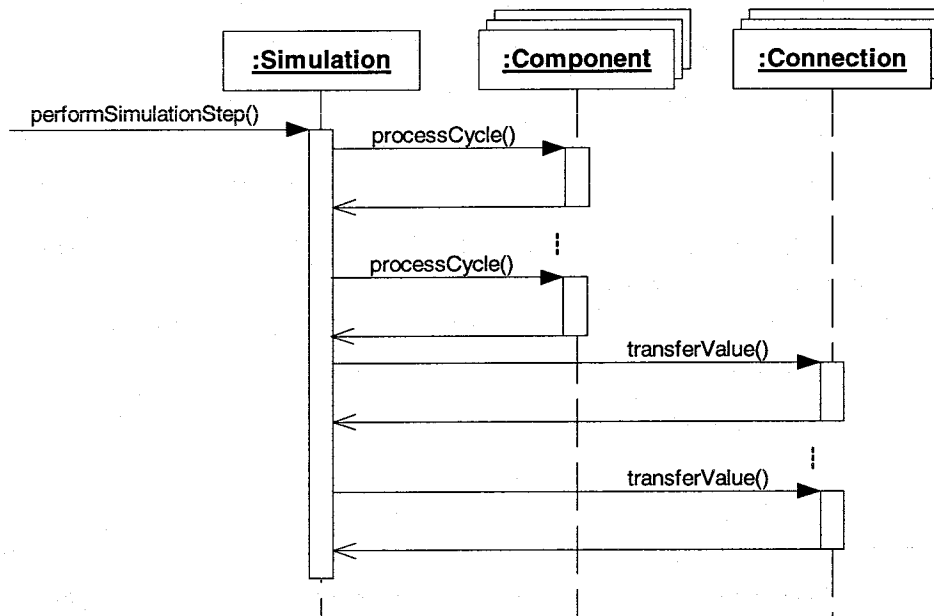


Figure 13: Sequence diagram of performSimulationStep method.

A simulation is derived from the class Simulation:

```

public abstract class Simulation
{
    private String name
    private int stepCounter
    private Collection components
    private Collection connections

    public Simulation(String name)
    public void performSimulationStep()
    public void addComponent(Component component)
    public void addConnection(Connection connection)
    public String getName()
    public int getStepCounter()
    public Collection getComponents()
}
  
```

}

The simulation class has four private variables:

- **name**: holds the name of the simulation,
- **stepCounter**: holds the current simulation step the simulation is in,
- **components**: a list of all components in the simulation,
- **connections**: a list of all connections in the simulation,

The constructor of a derived class must override the constructor of the base class. In this new constructor the structure that is to be simulated must be defined by adding components and connections to the simulation. Usually, the constructor is the only method that needs to be overridden.

The method `performSimulationStep` is implemented by two iterations: one iteration over the `components` collection in which the `performCycle` method of every component is called, followed by an iteration over the `connections` collection in which the `transferValue` method of every connection is called.

The other methods have very simple implementations. Method `addComponent` adds the component referred to by its only parameter to the collection `components`. Method `addConnection` adds the connection referred to by its only parameter to the collection `connections`. The methods `getName`, `getStepCounter` and `getComponents` return respectively the value of the string `name`, the integer `stepCounter` and the collection `components`.

2.2 Simulation GUI

To control the simulation, a graphical user interface has been created. The main control panel is shown in figure 14. It is a snapshot of the main control panel while it runs a simulation of the Image Blender.

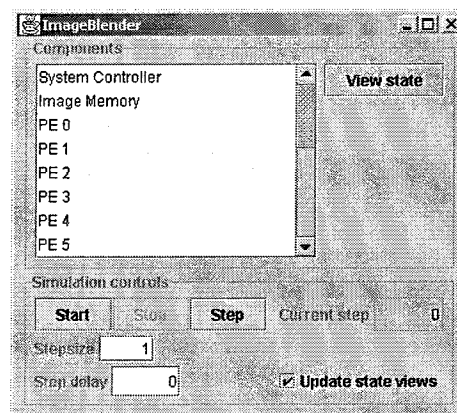


Figure 14: The simulation control window.

A list shows all components included in the simulation. The user can select components from this list and by pushing the 'View State' button a statewindow is opened for each selected component. An example of a statewindow is shown in figure 15. The statewindow shown is of a Processing Element from the Image Blender. It displays the state of each register of the PE.

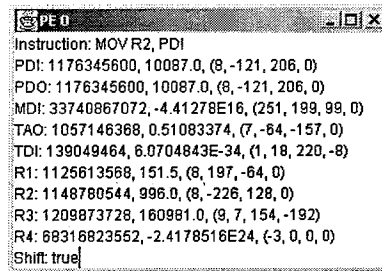


Figure 15: An example of a statewindow.

The simulation is started and stopped by pushing respectively the 'start' and the 'stop' button. A single simulation step can be performed by pushing the 'step' button. The 'current step' box displays the current step the simulator is in. The stepsize (i.e. the number of simulation cycles that is considered one simulation step) can be altered by changing the value in the 'stepsize' box. The simulation delay between each simulation step is defined by the value in the 'step delay' box. This delay is defined in milliseconds.

After each simulation step all opened statewindows are updated. This has a considerable impact on the simulation speed. By unchecking the 'update state views' checkbox the statewindows will only be updated when the simulation is stopped.

By default the state of a component is displayed using a textstring. However, for some types of components a different kind of state viewer would be desirable. These can be made by implementing a custom state viewer. To use these custom state viewers with the current simulation GUI, they have to implement the `ComponentStateViewer` interface. The new custom viewer then needs to be associated with the component it was intended for by calling the method `setStateViewer(component, viewer)`, where `component` is the intended component and `viewer` is the custom state viewer. `setStateViewer` is a public method of the current simulation GUI.

2.3 An example of a simulation

To demonstrate the simulation framework, in the next example two interacting buffers are simulated. Buffer 1 transfers values from its own bufferspace to buffer 2. As a whole, the system has no usefull function because the two buffers are not interacting with anything else. It merely serves as an example of how to implement a simulation.

First a Buffer component is defined by extending `Component` and defining its interface by adding connectors and its behavior by implementing the `performCycle` method.

```
class Buffer extends Component
{
    public IntConnector input = new IntConnector(-1);
    public IntConnector output = new IntConnector(-1);

    private int size, index;
    private int[] buffer;
```

```

public Buffer(String name, int size)
{
    super(name);
    this.size = size;
    index = 0;
    buffer = new int[size];
}

public void processCycle()
{
    output.setValue(buffer[index]);
    buffer[index] = (int) input.getValue();
    index = (index + 1)%size;
}
}

```

It is assumed that connectors of the type `IntConnector` are defined. The interface of the buffer component consists of an input connector `input` and an output connector `output`. Both are instances of the `IntConnector` class, which is assumed to be an existing class derived from the base class `Connector`, and which holds integer values. The behavior of the component is as follows: the buffer element referred to by `index` is copied to the output connector, the input value is put into the buffer and `index` is raised by 1.

Now a simulation can be built by extending `Simulation`:

```

public class Voorbeeld extends Simulation
{
    public static void main(String[] args)
    {
        new Voorbeeld();
    }

    public Voorbeeld()
    {
        super("Buffersimulation");

        try
        {
            Buffer buffer1 = new Buffer("Buffer 1", 6);
            addComponent(buffer1);
            Buffer buffer2 = new Buffer("Buffer 2", 2);
            addComponent(buffer2);
            addConnection(new BasicConnection(buffer1.output, buffer2.input));
        }
        catch (SimulatorException se)
        {
            System.exit(1);
        }
    }
}

```

```
    }  
  
    new SimulationGUI(this);  
  }  
}
```

In the above can be seen that two instances of `Buffer` are created and added to the simulation. One instance has the size 6 and is named 'Buffer 1'. The other has size 2 and is named 'Buffer 2'. They are linked together by a `BasicConnection`, which links the output connector of Buffer 1 to the input connector of Buffer 2. The simulation is started by creating an instance of `Voorbeeld`.

3 Implementation of two pass forward texture mapping on the Image Blender

The two pass forward texture mapping method as described in section 1.2.2 now needs to be implemented on the Image Blender. This is discussed in the next section. After this, the simulation of this implementation is described.

3.1 Two pass forward texture mapping on the Image Blender

The texture mapping process consists of the calculation of contributions of texels to pixels. These contributions need to be summed up for each pixel to obtain the final pixel values. If and how much each texel contributes to a certain pixel is determined with a filter function. The width of the used filter, F , is fixed. Combined with the fact that the algorithm deals with minification it is known that the number of pixels that a texel contributes to is known and equal to either F or $F + 1$, depending on the location of the texel borders. This is illustrated in figure 16.

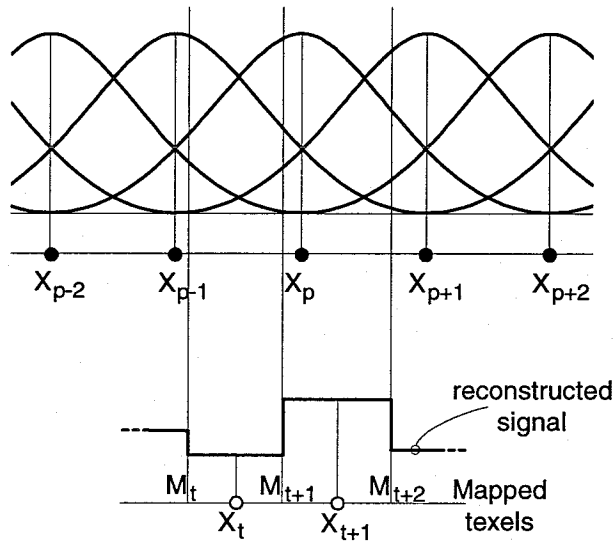


Figure 16: Contributions

In this figure F is equal to 4. Texel t contributes to pixels $p - 2$, $p - 1$, p and $p + 1$. The texel borders of texel t are both located between the pixel $p - 1$ and p . Texel $t + 1$ however has its texel borders on both sides of pixel p . Therefore it contributes to pixel $p - 2$ as well as pixel $p + 2$. So texel t and texel $t + 1$ contribute to respectively F and $F + 1$ pixels.

3.1.1 Distribution of the texel processing

There are two ways to distribute texels over the PE's of the Image Blender:

- *Every PE processes the same texel.*

Each PE receives the same texel value C_t and determines how much this texel contributes to a pixel by calculating the contribution factor for this pixel. C_t is scaled using this factor and the result is added to the sum of the previous texel contributions.

When the last texel contribution to a pixel is added, the resulting pixel is finished. These results are represented by C_p , C_q and C_r .

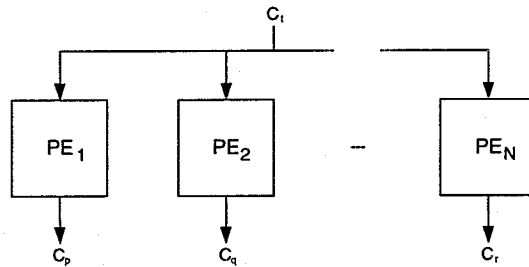


Figure 17: Every PE processes the same texel.

This implementation fits a synchronous (each instruction is executed on all PE's at the same time) as well as a delayed (each PE executes an instruction one cycle after the previous PE) execution order. However, because the Sample Mapper is meant to deliver a block of subsequent values it needs to be adapted to deliver the same texel value to multiple PE's.

Because each PE calculates one contribution of a texel, this implementation can only be used when the number of PE's is equal to F , in order to avoid idle PE's. When the number of PE's is a multiple of the filterwidth, the structure can be repeated multiple times to form subsets of PE's that process a texel. However, the texel values that serve as input for each separate subset have to come from different scanlines, otherwise it would be possible that the calculation in one subset depends on contribution values only available in another subset. This implies that there will also be results for different scanlines, so multiple writebacks to the memory are needed.

- *Every PE processes a different texel.*

Every PE receives a subsequent texel value. It calculates the contributions of this texel to the different pixels by calculating the contribution factors and scaling the texel appropriately. These contributions are added to the contributions to the same pixels received from the previous PE. The results are sent to the next PE. If the added contribution is the last contribution to be added to the pixel, the pixel is finished. These final pixel results are represented by C_p , C_q and C_r . A datapath between PE's is needed to transfer contributions to the next PE.

This implementation only fits a delayed execution order because the sum of the contribution values depends on contributions calculated in previous PE's. It scales nicely for any amount of PE's and also fits the way the Sample Mapper delivers texel values to the PE's, i.e. as a block of subsequent texels.

In both methods the same amount of contributions need to be calculated, so only the distribution of these calculations over the PE's is different. Therefore both require an equal amount of operations. Because the Image Blender already contains datapaths between PE's

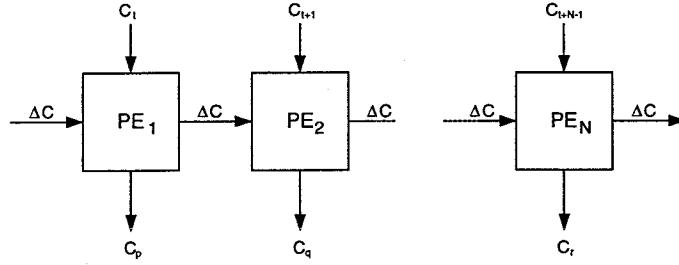


Figure 18: Every PE processes a different texel.

and the fact that the latter method scales nicely and does not need any modifications to the Sample Mapper, the latter method is chosen. This also implies that a delayed execution order is chosen.

3.1.2 Resampling equation

The resampling equation 3 from section 1.2.2 is restated below:

$$C_p = \sum_{t=b}^e (H(M_{t+1} - X_p) - H(M_t - X_p)) C_t \quad (5)$$

Every PE receives a texel value C_t . For every pixel p to which texel t contributes, the contribution factor $H(M_{t+1} - X_p) - H(M_t - X_p)$ needs to be calculated. A lookup table is used to store the values for H , so 2 table lookups are needed for the calculation of a single contribution factor. With the contribution factor the value C_t is scaled. This scaled value is added to the sum of the contributions of previous texels to this pixel received from the previous PE and the result is sent to the next PE.

The number of contributions that needs to be calculated is equal to either F or $F + 1$. However, because all the PE's execute the same instructions each PE has to calculate $F + 1$ contributions.

Because of this each PE needs to perform $2(F + 1)$ table lookups and $F + 1$ texel scalings for every texel. Equation 5 can be rewritten [Meinds00b] in such a way that it requires fewer operations.

$$C_p = C_e + \sum_{t=b+1}^e H(M_t - X_p) (C_{t-1} - C_t) \quad (6)$$

When this equation is used, every PE not only needs a texel value C_t , but also the value of the previous texel, C_{t-1} . Since the PE's calculate the contributions of subsequent texels this is not a problem, the texel values can be shared between PE's.

First ΔC is calculated by subtracting C_t from C_{t-1} . Then for every pixel p to which texel t contributes, the contribution factor $H(M_t - X_p)$ is calculated. With this contribution factor ΔC is scaled, then added to the value for this pixel received from the previous PE and the result is sent to the next PE.

Again, a texel t contributes to either F or $F + 1$ pixels. However, when equation 6 is used and the last texel e to a certain pixel p is being processed, the needed $F + 1$ pixel contributions consist of F scaled ΔC values and one unaltered C_e value. Therefore in both cases only F pixel contributions need to be calculated. So, every PE performs exactly F times the calculation of a contribution factor followed by a texel scaling, instead of the $F + 1$ times when equation 5 is used. Therefore it only needs F table lookups and F texel scalings.

Because equation 6 requires less than half of the table lookups and saves one texel scaling when compared to equation 5, equation 6 is chosen over equation 5.

3.1.3 Processing the texels

A texel t contributes to the pixels p_1, \dots, p_F , and depending on the position of the texel borders also to pixel p_{F+1} . The contribution ΔC_{p_i} of texel t to pixel p_i is calculated following equation 6, so $\Delta C_{p_i} = H(M_t - X_{p_i})(C_{t-1} - C_t)$. These contributions are added to the contributions of previous texels to the same pixel, calculated by the previous PE's.

The processing of a texel t consists of the following steps:

- 1 Calculate the difference between texel $t - 1$ and texel t : $\Delta C_t = C_{t-1} - C_t$
- 2 Determine if t is the last texel that contributes to pixel p_1 : $\text{shift} = \text{LSB}(I(M_t)) \text{ XOR } \text{LSB}(I(M_{t+1}))$
- 3 Determine the contribution factors: $f = H(\text{FractionalPart}(M_t))$
- 4 Calculate the contributions:
 - 4.1 $\Delta C_{p_1} = f_1 * \Delta C_t$
 - ⋮
 - 4.F $\Delta C_{p_F} = f_F * \Delta C_t$
- 5 Add the contributions to the contributions of previous texels and forward the results to the next PE. While the order in the previous step may be chosen arbitrarily, the order in this step is very important:
 - 5.1 if shift then $C_{out_F} = 0$
else $C_{out_F} = \Delta C_{in_F} + \Delta C_{p_F}$
 - ⋮
 - 5.i if shift then $C_{out_i} = \Delta C_{in_{i+1}} + \Delta C_{p_{i+1}}$
else $C_{out_i} = \Delta C_{in_i} + \Delta C_{p_i}$
 - ⋮
 - 5.F if shift then $C_{out_1} = \Delta C_{in_2} + \Delta C_{p_2}$
else $C_{out_1} = \Delta C_{in_1} + \Delta C_{p_1}$
- 6 if this is the last texel to be added to pixel p_1 , calculate the final value for this pixel:
if shift then $C_p = \Delta C_{in_1} + \Delta C_{p_1} + C_t$

Step 1: calculate the difference between texel $t - 1$ and texel t .

In the calculation of ΔC_{p_i} , $(C_{t-1} - C_t)$ is the same for every contribution of the same texel t , so this has to be calculated only once: $\Delta C_t = C_{t-1} - C_t$. The subtraction in this equation is a color component wise subtraction, i.e. the red component of C_t is subtracted from the red component of C_{t-1} , the green component of C_t from the green component of C_{t-1} , etc.

Step 2: determine if t is the last texel that contributes to pixel p_1 .

To determine if texel t is the last texel that contributes to pixel p_1 , M_t and M_{t+1} , i.e. respectively the left and right border of texel t , need to be compared. This can be seen in figure 19. In this figure $F = 4$ and the borders of texel t , i.e. M_t and M_{t+1} , fall each on one side of pixel p_3 . Combined with the fact that pixels lie on an integer grid, this implies that the integer part of M_{t+1} is larger than the integer part of M_t . In fact, when $I(x)$ defines the integer part of x then $I(M_{t+1}) = I(M_t) + 1$. So $I(M_{t+1})$ and $I(M_t)$ will only differ in the least significant bit (LSB) of both values, and when these are different, texel t is the last texel contributing to pixel p_1 .

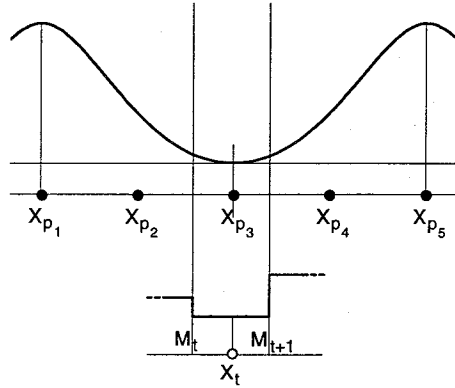


Figure 19: Texel t is the last texel that contributes to pixel p_1 .

Step 3: determine the contribution factors.

Each contribution ΔC_{p_i} is calculated with $\Delta C_{p_i} = H(M_t - X_{p_i})\Delta C_t$. Here the value ΔC_t is scaled by the contribution factor f_i , with $f_i = H(M_t - X_{p_i})$. Again, each color component of ΔC_t is scaled separately.

A table lookup in the H table is needed for the determination of each contribution factor. Between texels there is no regular pattern in the lookup actions, which would have allowed the delivery of a block of subsequent values from the H table using the Image Memory and the Sample Mapper. Therefore the Lookup Table from the Image Blender has to be used to host the H table. Multiple lookup tables are necessary to allow for the required amount of lookup actions.

Luckily there exists a regular pattern in the lookup actions of one single texel, which allows the combination of all the table lookup actions for one texel into one lookup action [Meinds-Peters01]. This is based on the fact that all pixels lie on an integer grid and hence it is known that $X_{p+i} = X_p + i$. When this is applied to the calculation of the contribution factor, i.e. $H(M_t - X_{p+i})$, this leads to $H(M_t - X_p - i)$. This means that all lookup actions for one texel can be based on one base index $M_t - X_p$ and all the others are integer increments/decrements of this base index.

If the base index is $\varphi = M_t - X_p$, where X_p is the position of the pixel closest to the right of M_t , then table H can be organized in such a way that it consists of a row for every different value of φ . Each row is indexed with φ and contains the F contribution factors for one texel, i.e. the values $\dots, H(\varphi - 1), H(\varphi), H(\varphi + 1), \dots$. Also, the value of φ will be in the range $[0..1]$, so it can be obtained by taking the fractional part of M_t .

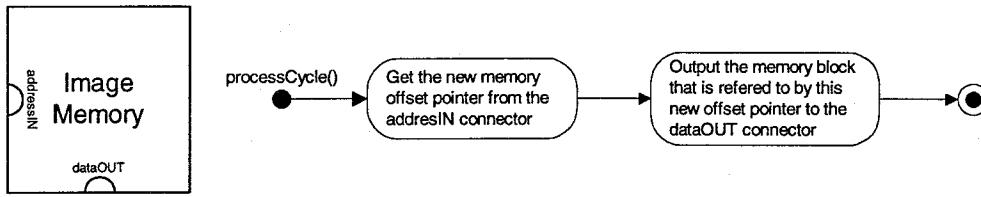


Figure 20: Schematic representation and activity diagram of the Image Memory component.

- dataIN: for receiving data belonging to an instruction,
- instrOUT: to output the instruction for the next PE,
- dataOUT: to output data belonging to the instruction for the next PE,
- memoryIN: for receiving data from the Image Memory,
- tableOUT: for selecting an element in the Lookup Table,
- tableIN: for receiving data from the Lookup Table.

Note that the memoryOUT connector is missing. The output pixels are captured directly, and written to a file. This way the result can be compared to the result of a reference implementation to test the correctness. The simulation of the second pass is almost identical to the first pass. Though, it has to be initialized with the result from the previous pass, i.e. the intermediate image instead of the source texture.

In each cycle the instruction received on instrIN is copied to instrOUT. Depending on the instruction, the data from dataIN is also copied to dataOUT. Connectors are accessed like registers, so registers are used to simulate this. Therefore the values of the input connectors are copied to their respective registers (pdi, mdi, tdi) from which input data can be accessed. After this the instruction is decoded and executed. When the execution is finished, the values of the output connectors are updated with the values from their respective registers (pdo, tao).

Figure 21 shows a schematic representation of a Processing Element, together with a diagram of its behavior.

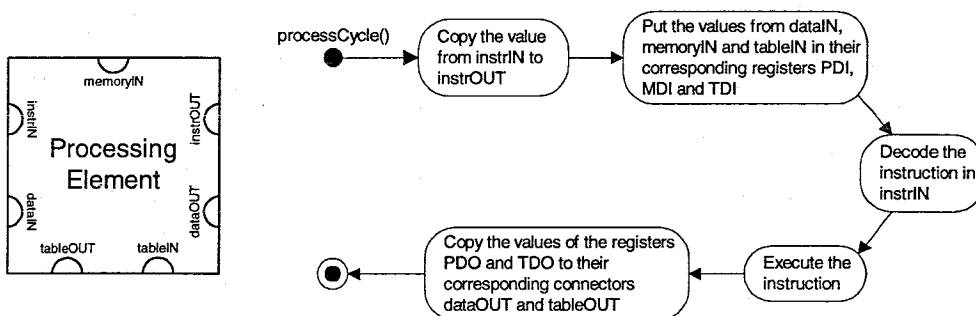


Figure 21: Schematic representation and activity diagram of the Processing Element component.

3.2.3 Lookup Table

The Lookup Table basically works the same as the Image Memory. It also has a connector to select the offset in the table, called indexIN, and a connector to output the selected data, called dataOUT.

A diagram of the behavior of a Lookup Table component, as well as a schematic representation, can be found in figure 22.



Figure 22: Schematic representation and activity diagram of the Lookup Table component.

3.2.4 System Controller

The System Controller controls the entire system. The interface consists of:

- instructionOUT: to issue the instructions to the first PE in the array,
- dataOUT: to send accompanying data with the instructions,
- dataIN: to receive data from the last PE in the array,
- addressOUT: for controlling the Image Memory,
- globaldataOUT: to send data to all PE's via the Sample Mapper.

The System Controller sends an offset value to the Image Memory to select a certain block of data. One cycle later this block is available to the Sample Mapper. By sending a data value to the Sample Mapper, the entire array of PE's can be provided with the same value.

The behavior of the System Controller is divided into steps. Every cycle one step is performed. Each step issues one instruction of the program in sections 3.1.5 and 3.1.6. After one loop is completed the process starts over. This loop is continued until the entire texture is processed. A schematic representation of the System Controller is shown in figure 23.

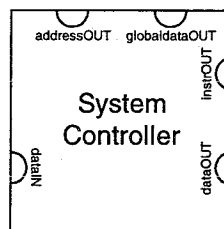


Figure 23: Schematic representation of the System Controller component.

3.2.5 Sample Mapper

The Sample Mapper is implemented by a connection. This is done under the assumption that the Sample Mapper performs the delivery of values without any delays. It distributes a row of

memory elements from the dataOUT connector of the Image Memory to the memoryIN connectors of the PE's, while introducing the appropriate delays. These delays are implemented as shown in figure 8. It also distributes a value received from the globaldataOUT connector of the System Controller, and the same delay is introduced.

3.2.6 Connections

The Image Memory and the PE's are linked together by the SampleMapper. The Lookup Table is connected to the PE's with a shared bus. All other connections are instances of BasicConnection. The complete structure is shown in figure 24.

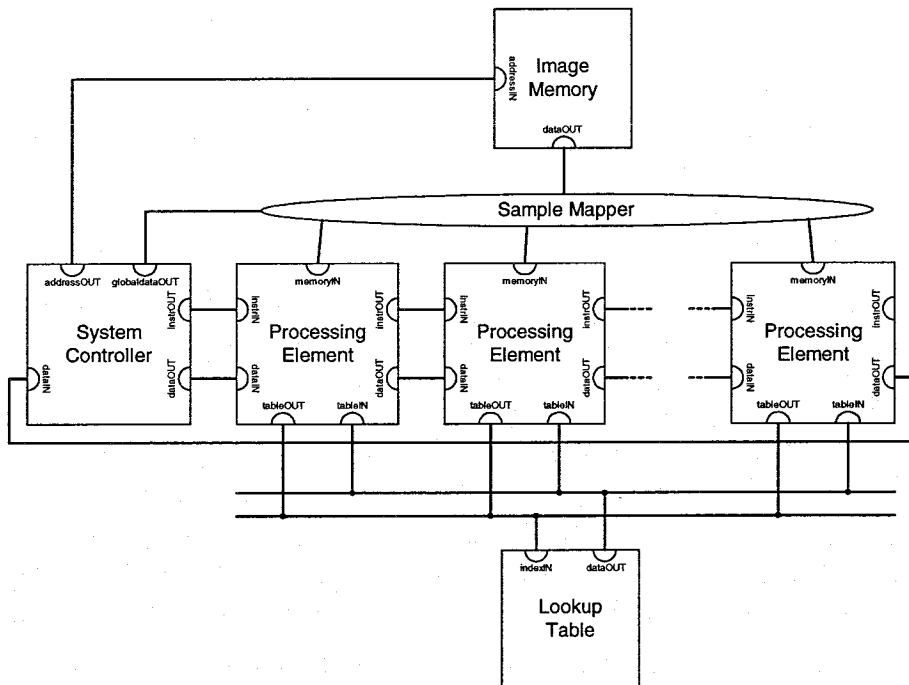


Figure 24: The complete structure.

4 Performance comparison

The number of cycles needed to process a single texel can be derived from the program in section 3.1.5, and equals $8 + F$. If there are N PE's available, then the processing of one pass for a texture image consisting of Y scanlines of X texels takes $Y(X/N)(8 + F)$ cycles. If X is not a multiple of N then this amount needs to be increased with $Y(8 + F)$. Also, the final result is available N cycles later because of the delayed execution method. This delay is however unimportant if there are more textures to be processed.

To compare the performance of two pass forward texture mapping on the Image Blender with a pure hardware implementation, the performance of the transposed polyphase structure from [Meinds00b] needs to be determined. This structure utilizes F multipliers to scale texel values. It is capable of processing a texel in one cycle.

To make a fair comparison, the hardware structure is compared to an Image Blender with F PE's, because the hardware structure has F multipliers at its disposal, and a PE is assumed to have only one. Also, the part of the texel processing program which calculates the M_{t+1} values is excluded from the comparison, since the hardware structure does not calculate these values by itself. This reduces the number of cycles needed to process a single texel to $3 + F$. This number can be further decreased by combining steps. However, the minimum number of cycles will be $F + 1$ because the TM3, TM4 and TM5 operations all use the multiplier, so without adding multipliers these steps can not be combined. So the difference between both methods is the 1 to 3 cycles overhead.

Therefore implementation of two pass forward texture mapping on the Image Blender is, as was to be expected, less efficient than the hardware implementation. However, the difference is not that large, especially for larger values of F . And by using the Image Blender to do the mapping then there is no need for an extra dedicated module, so the entire system is more flexible.

5 Further work

In section 3.1.6 a module is mentioned that puts the finished pixel values back into the Image Memory. The implementation of such a module must be further studied. One possibility is to shift the received values into a buffer, and copy the values into the Image Memory when the buffer is full. It is also possible to have the finished pixel values put onto a shared bus for external output.

Another issue is the calculation of the texel borders. In this calculation each PE uses a division operation. This might be too costly. Therefore it must be researched if it is feasible to replace this division with a midpoint algorithm, i.e. replace the division by an approximation algorithm that uses only integer arithmetic.

It might be impossible to implement an efficient midpoint algorithm on the Image Blender. The midpoint algorithm tries to maintain an invariant by incrementing or decrementing variables, depending on how the invariant was violated. These variables are incremented or decremented using a guarded loop until the desired value is reached. The number of necessary iterations of the loop varies. In an SIMD architecture however, every PE executes the same

instructions. So every PE has to perform the same amount of iterations. This severely decreases efficiency.

Also, to be more efficient, the midpoint algorithm depends heavily on previous results. These are used to estimate the final result so the number of iterations can be restricted. On the Image Blender however, multiple values are calculated at the same time. Therefore previous results can not be used since these values are still being calculated by the other PE's. So the end result can not be estimated and this results in a very high number of iterations.

6 Conclusion

In this report the implementation of a two pass forward texture mapping algorithm on the Image Blender SIMD processing architecture has been discussed. The implementation is simulated to test the correctness, find problems and estimate the performance.

To perform two pass forward texture mapping on the Image Blender, each Processing Element (PE) calculates the contributions of one texel from a block of subsequent texel values. These contributions are shared between PE's and by adding them together the final pixel values are calculated.

The algorithm offers great flexibility as opposed to the combination of a separate dedicated texture mapping unit with the Image Blender, because the dedicated hardware can not be used for other tasks. Also, it scales very well, so the amount of PE's can be chosen arbitrarily. However, when the number of PE's is larger then the number of cycles needed to process a texel, extra lookup tables will be needed to provide for the increased number of table lookups per cycle.

Performing two pass forward texture mapping on the Image Blender is, as was to be expected, less efficient than a dedicated hardware implementation. However the difference in the number of cycles is not that large, so the scalability and flexibility might be more important.

The performance can be improved by using mipmapping, i.e. extending the original texture with a set of downsampled versions of this texture with decreasing resolution. However, apart from the inherent difficulties of mipmapping in combination with two pass forward texture mapping, the combination with the Image Blender introduces a new efficiency problem. Because all PE's receive texel values in a block of subsequent texels, all values must come from the same mipmap level. This limits the effectiveness of mipmapping, especially when the number of PE's is high.

The algorithm is simulated using a component oriented simulation framework. This framework is based on components, which have connectors and are connected via these connectors using connections.

The simulation of the Image Blender is successfully implemented using this framework. All the separate parts that make up the Image Blender are implemented as components. Each separate component is tested individually for correct operation with the using stubs, i.e. a component with an implementation that is just sufficient to test the component in question. Then all the components are connected together to form a complete system.

The simulation has been mainly used to find problems in the algorithm and to prove that the algorithm works correctly. The performance however is calculated using analytical methods, although it could also be derived from the simulation.

The simulation framework is limited to simulating the behavior of a component to the level of one simulation cycle, i.e. a simulation cycle is the finest unit of time in the simulation. Therefore it can not be used to simulate the timing of an operation inside such a cycle.

By extending the implementation, most importantly the addition of instructions on the PE, the simulator can be used to test other algorithms as well. However, this has not yet been done.

References

- [Barenbrug00] Bart Barenbrug, "Midpoint-based division free perspective-correct texture mapping.", Nat.Lab. Technical Note 2000/404
- [Catmull-Smith80] Ed Catmull and Alvy Ray Smith, "3-D transformations of images in scanline order", Computer Graphics SIGGRAPH proceedings 1980
- [Kuijk00] A.A.M Kuijk, "The Image Blender: a SIMD coprocessor architecture for pixel based image synthesis", Nat.Lab. Technical Note 2001/531
- [Meinds00a] Koen Meinds, "Perspective transformation of images with a FIR filter", Nat.Lab. Technical Note 2000/141
- [Meinds00b] Koen Meinds, "Variable sample rate conversion without DC-ripple", Nat.Lab. Technical Note 2000/422
- [Meinds-Peters01] K. Meinds and F.J. Peters, "Texture mapping with digital video filters", 2001