Implementation of a spatio-temporal Laplacian image pyramid on the GPU

Sönke Ludwig

February 5, 2008

Bachelorarbeit im Studiengang Informatik
am Institut für Neuro- und Bioinformatik,
Technisch-Naturwissenschaftliche Fakultät der Universität zu Lübeck,
Ausgegeben von PD Dr.-Ing. Erhardt Barth
Betreut durch Dipl.-Inf. Michael Dorr

Abstract

This thesis presents an implementation of Laplacian image pyramids in the spatial and temporal domains of a video stream. All of the computationally expensive per-pixel work has been implemented on the graphics card, using Cg fragment shaders.

The high vector processing performance even of cheap mid-range graphics cards outperforms the CPU by far for many imaging tasks. Using this hardware, it is possible to perform full spatio-temporal filtering of high resolution video material at interactive framerates beyond 30 fps.

This hardware accelerated filtering system has been integrated into an existing CPU based framework to allow for nearly seemless switching and mixing of CPU and GPU based algorithms, as well as to allow for straightforward implementations of further GPU based algorithms.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Lübeck, den 5. Februar 2008

.....
Sönke Ludwig

Contents

1 Introduction

The Gaussian image pyramid is a simple multi-resolution pyramid commonly used for image processing. It stores a set of successively lowpass filtered and subsampled versions of an image to allow for fast access to image features of specific frequencies or sizes. The Laplacian pyramid is an extension to the Gaussian pyramid and was first described by P. J. Burt and E. H. Adelson in [1] and [4]. The original use for this multi-resolution pyramid was data compression. In the pyramid, an image is seperated into distinct frequency bands. This reduces the correlation of neighbouring pixel values and largely reduces the number of non-zero pixels. The resulting – mostly zero and reduced in resolution – images provide a good basis for various compression algorithms.

Because of the distinct frequency bands stored in the pyramid, it is also very well suited for frequency or size dependent filtering and image analysis. Such techniques include image blur and sharpen filters, efficient template matching [3], image mosaicing [5], and texture synthesis [9].

Also, various research has been done in the area of human vision, where Gaussian pyramids were used for frequency filtering. This research includes the simulation of visual fields and investigation of the effects of frequency filtered video streams on the eye movements of the observer (See [8], [6], [2]). Spatial and temporal filtering of videos has been employed. The algorithms which were used in these papers are CPU based and can easily push typical PC processors to their limits, given a sufficiently large video image resolution. Laplacian pyramids could not be used because of their even higher computational requirements.

Over the last years, the shading capabilities of graphics cards have seen increasing uses in general purpose computation, because of their high floating-point performance. Also, the flexibility of the shader programs, which are used for those computations, has recently become almost identical to that of a CPU (missing mostly pointer-arithmetic). Using this processing power, it is now possible to transform a large set of algorithms to use the vector processing units of the graphics card.

A simple form of a hardware accelerated spatial Gaussian image pyramid is described in [7]. This approach utilized the texturing capabilities of (non-shader capable) graphics cards to implement an approximation to a Gaussian pyramid, which was used to simulate visual fields.

With the use of todays shader capabilities, we can go some steps further

now and implement a full GPU[1]-based Laplacian pyramid in the spatial domain, as well as the temporal domain of a video stream. The performance of current mid-range hardware allows for full filtering of high definition videos (1280x720) at interactive frame rates beyond 30 Hz. This implementation has been integrated into an existing framework used for realtime gaze-contingent manipulation of videos using an eye-tracker.

2 The Laplacian Pyramid

2.1 Gaussian Pyramid

The Gaussian pyramid is the base structure from which later the Laplacian pyramid is built. It contains a stack of signals (images in the spatial case), where each consecutive level contains a successively lowpass filtered version of the original signal. The lowpass filtered signals can be stored at reduced sampling rate an thus require less storage space.

The pyramid is constructed by iteratively applying a lowpass filter and subsampling of the resulting signal. This process is repeated until a signal length of 1 or a specific maximum number of levels is reached. Because ideal lowpass filtering is expensive, a Gaussian filter is used as an approximation (5-tap binomial kernel). The upper part of Figure 1 shows the pyramid generation in the spatial case.

With this structure, it is now possible to generate arbitrary approximated lowpass filtered versions with very low computational overhead. The construction of these images is done by interpolating between neighbouring levels. The cutoff frequency can be varied per pixel by choosing different interpolation coefficients and levels.

Using this property to efficiently generate arbitrary lowpass filtered versions of the individual pixels, gaze-contingent displays can be implemented, which vary the amount of information (cutoff frequency) of the displayed image per-pixel, depending on the gaze direction of the observer. [8] gives a detailed description of such a system.

[1]GPU = Graphics Processing Unit

2.2 Laplacian Pyramid

The Laplacian pyramid is an extension of the Gaussian pyramid. It stores distinct frequency bands instead of only lowpass filtered signals. This allows the original signal not only to be lowpass filtered, but it becomes possible to amplify or attenuate the frequency bands individually.

To build the extended pyramid, for each level of the Gaussian pyramid, the difference of the Gaussian signal with the lower resolution level is computed. This difference corresponds to a bandpass filtered version of this level. The result is a set of bandpass filtered images which form the basis functions for reconstructing or modifying the original signal (See Figure 2).

Because of the different sampling rates of the levels, in each step, the low resolution level has to be upsampled by a factor of two before the substraction can be performed. This upsampling procedure works by first taking the low resolution signal, inserting zeros between every neighbouring samples, thus doubling the signal resolution, and then applying the same 5-tap binomial kernel, which was used in the downsampling step. This results in a smooth, position invariant interpolation without discontinuities in the upsampled result.

The image can be reconstructed perfectly by reversing the construction steps. The lowest resolution level is taken, upsampled, and added pixel by pixel to the next higher level, resulting in the correponding Gaussian level. This process is then repeated until the highest resolution level is reached and the image is reconstructed. See Figure 1 for a breakdown of the whole algorithm. For changing the contribution of the frequency bands, a scale factor can be applied on each iteration of the reconstruction algorithm to weight the levels differently.

2.3 Spatio-temporal application

In the case of this thesis, the signal is a video stream which is to be filtered in both the two spatial and the temporal dimensions. This is achieved by first creating a pyramid for only the temporal dimension. Then, after reconstructing a modified version from this temporal pyramid, a spatial pyramid is built from the output image. The up- and downsampling for the spatial pyramid is done simultaneously in both dimensions of the image (see Figure 1).

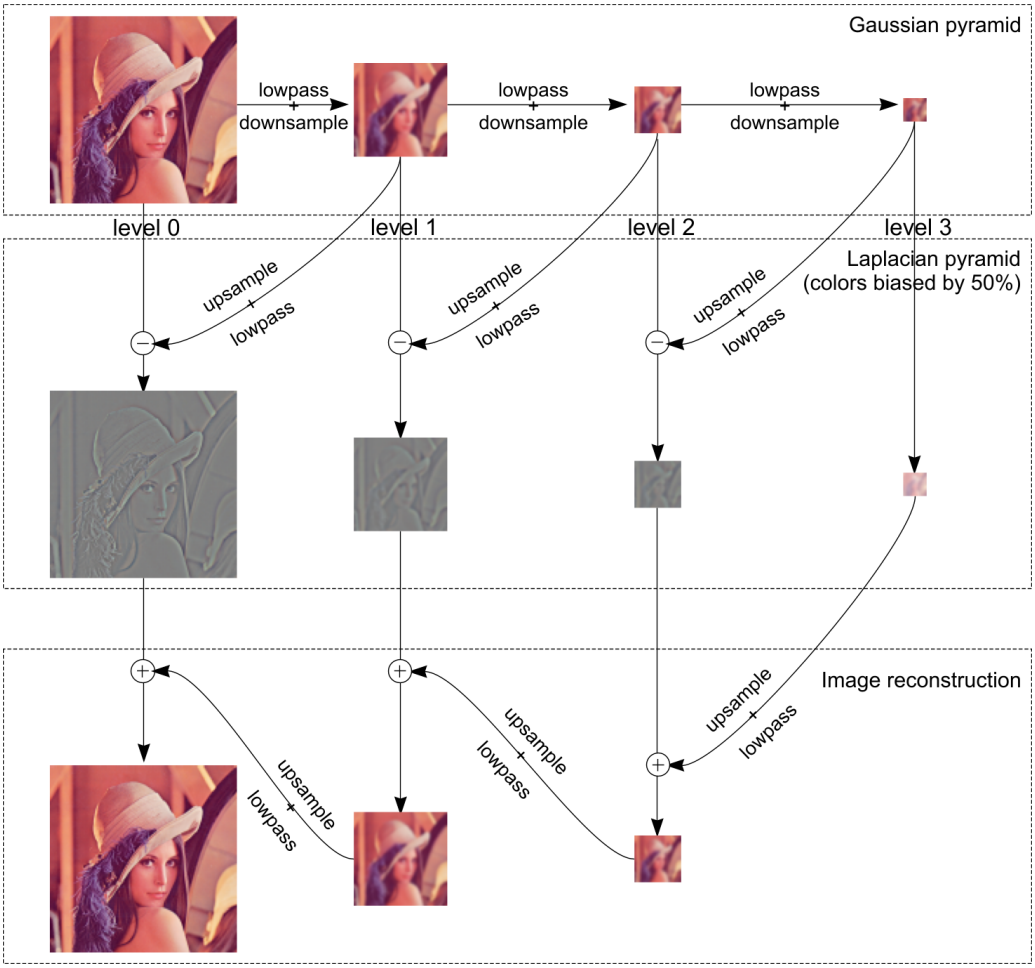


Figure 1: Generation and reconstruction of the spatial Laplacian pyramid

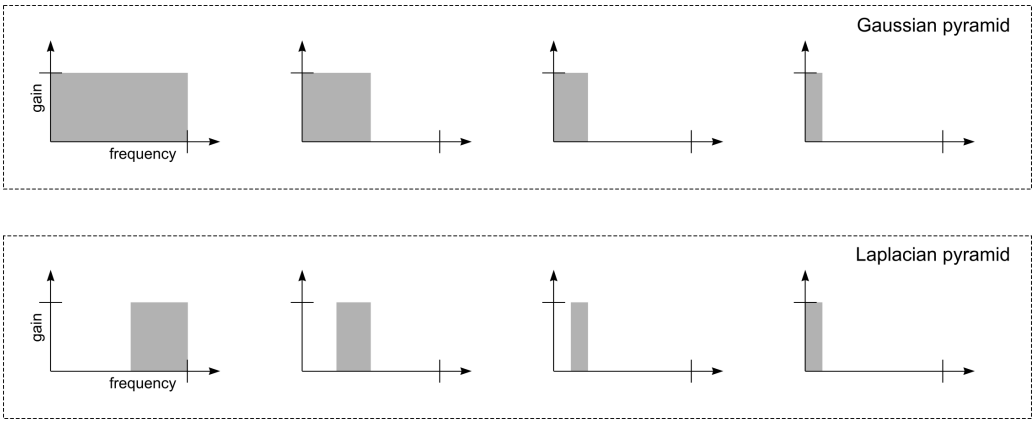


Figure 2: Approximated spectrum of the images stored in the pyramid

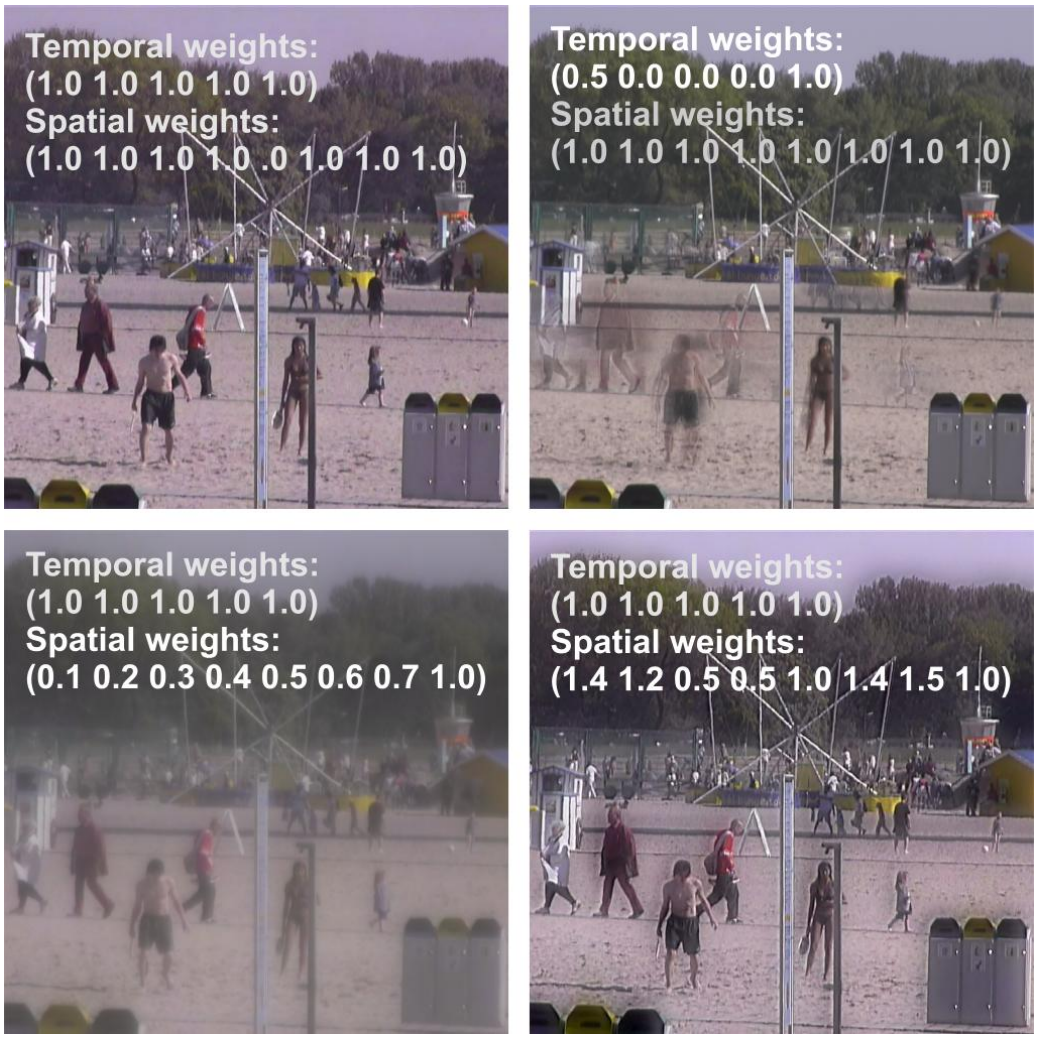


Figure 3: Examples of different Laplace reconstruction weights
The tempo-spatial pyramid used for these images has eight spatial levels and five temporal levels. Weights are given from highest to lowest resolution level.

3 The OpenGL/Cg Rendering Pipeline

This section lists the structure of the rendering pipeline, on which the framework is built. See Figure 4 for a simple graphical overview. The basic stages of the pipeline are:

1. Vertex processing

   At this stage the vertices, which make up the render primitives (such as points, lines, triangles), are transformed into clip-space. Additionally, arbitrary computations can be done on each vertex using a vertex shader. Such a vertex shader gets a set of per-vertex attributes, such as position and texture coordinates, as well as user defined constant values ('uniforms'). It can then compute the clip-space vertex position and optional user-defined values ('varyings'), which will be interpolated across the primitive and are available as inputs to the fragment shader.

2. Primitive assembly

   Here, the vertices of the previous stage are used to build up the actual geometric primitives (e.g. triangles). In the case of image filtering, Those primitives will always be pairs of triangles, forming rectangles which fill the whole rendering area.

3. Rasterization

   The primitives are now typically rasterized using a scanline algorithm. A fragment shader can be used to modify the output for each pixel (or more specific 'fragment'). The final fragments are then written either to a framebuffer, which is displayed to the user, or to a texture, which can be used for further rendering.

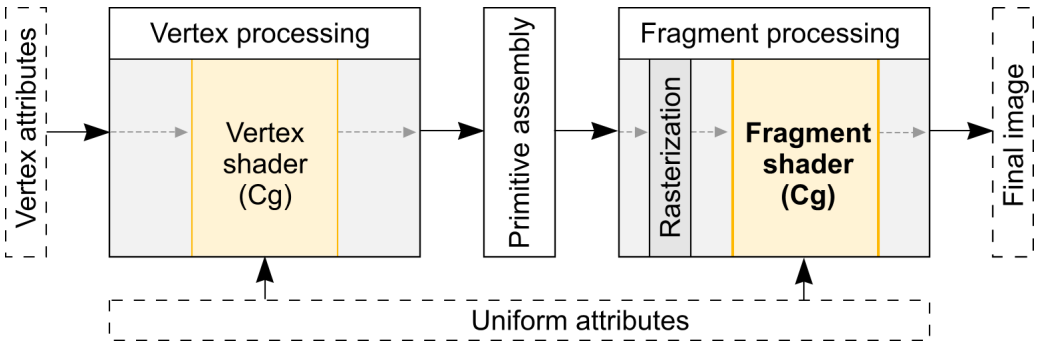


Figure 4: Overview of the OpenGL graphics pipeline

In the system implemented for this thesis, only rectangles are drawn, which cover the whole render target and the only interesting stage is rasterization. The z-coordinate of is set to zero, so, effectively, only the x and y dimensions of the 3D pipeline are used. The vertex processing stage will just automatically transform the vertices into clip-space and interpolate the texture coordinates over the rectangles. The fragment shaders are responsible for all of the important work.

The fragment shader will receive, in addition to the texture coordinate, an arbitrary set of textures and floating point values as inputs (uniform attributes). The input textures can be sampled at any coordinate and any number of times up to a hardware-dependent limit. This allows for a large number of filters to be implemented directly as a shader.

4 Implementation on the GPU

4.1 Framework

The communication with the GPU is done here using Cg[2] fragment shaders on OpenGL[3]. The platform specific code is hidden behind a set of classes, which provide a high level interface for performing texture operations. In Listing 1, pseudo-code is given for a simple application, which applies spatio-temporal filtering to a video stream using constant weights.

The system has been designed to fit into the already existing CPU based framework. It is possible to mix CPU and GPU based algorithms by interchanging images, which are stored in system RAM. The abstractions of the system have been chosen to allow for simple implementation of further GPU based algorithms.

The following enumeration lists the classes along with the most important methods, which are used in this paper. Figure 5 gives a graphical overview of the classes.

GLTextureWindow:

The window used to display a single texture image stretched over the full window area. A shader can optionally be applied for output modifications such as colour space conversions.

- SetTexture(texture): Sets the displayed texture
- SetShader(shader): Sets the shader, that is applied to the displayed texture

Texture:

A basic (two dimensional) texture image. Textures are stored on the graphics card and are either initialized using existing image data, or by using a TextureTarget/TextureOperator.

- SetImageData(image): Updates the texture with the specified image data from system RAM.

[2] `http://developer.nvidia.com/page/cg_main.html`
[3] `http://www.opengl.org/`

Shader:

A full shader program (fragment shader and optional vertex shader linked together). The shader program is loaded from an external text file stored on the file system. Parameters can be specified using the SetInput(...) methods. Textures and floating point values are allowed as parameters.

- SetInput(name, value): Sets the specified uniform input (texture or floating point value).

TextureTarget:

Performs render-to-texture operations using an FBO[4]. This class is a helper class used by TextureOperator.

- Setup(width, height): Sets the size of the destination texture.
- BeginRender(): Starts the rendering process.
- FinishRender(): Finalizes the rendering and stores the resulting image into an internal texture.
- GetTexture(): Returns a reference to the internal destination texture.

TextureOperator:

Applies a shader to a given set of input values. Input values can either be scalar values or texture images. The resulting image is stored as a target texture, which can in turn be used for successive processing.

- Setup(width, height): Sets the size of the destination texture.
- SetShader(shader): Sets the shader, which is used to process the inputs.
- SetInput(name, value): Sets the specified input for the currently set shader.
- Execute(): Performs the rendering into an internal destination texture using the specified shader and input values.
- GetResult(): Returns a reference to the internal destination texture.

[4] `http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt`

TextureInterpolator:

Performs a simple weighted sum operation on a set of textures. The resulting image is again stored into a texture. If a texture is specified more than once, its associated weights will be added and the texture is only accessed once.

- Setup(width, height): Sets the size of the destination texture.
- BeginInterpolation(): Starts a new interpolation process.
- AddImage(texture, weight): Adds the specified image, multiplied by 'weight'. Negative values are allowed.
- FinishInterpolation(): Finalizes the interpolation operation and returns a reference to the resulting texture.

SpatialPyramidCG:

Implements the image space (spatial) Laplacian pyramid.

- Setup(width, height, levels): Initializes the size of the pyramid.
- SetImageData(texture): Sets the lowest level (full resolution) and updates the rest of the pyramid accordingly.
- GetLaplace(weights): Reconstructs an image using the specified weights.

TemporalPyramidCG:

Implements the animation space (temporal) Laplacian pyramid.

- Setup(width, height, levels): Initializes the size of the pyramid.
- AddFrame(texture): Adds a new image to the sequence of the lowest level and updates the pyramid.
- GetLaplace(weights): Reconstructs an image using the specified weights.

Only GLWindow, Texture, Shader and TextureTarget contain platform specific code. This makes porting to different platforms and shading languages (such as Direct3D/HLSL or OpenGL/GLSL) an easy task.

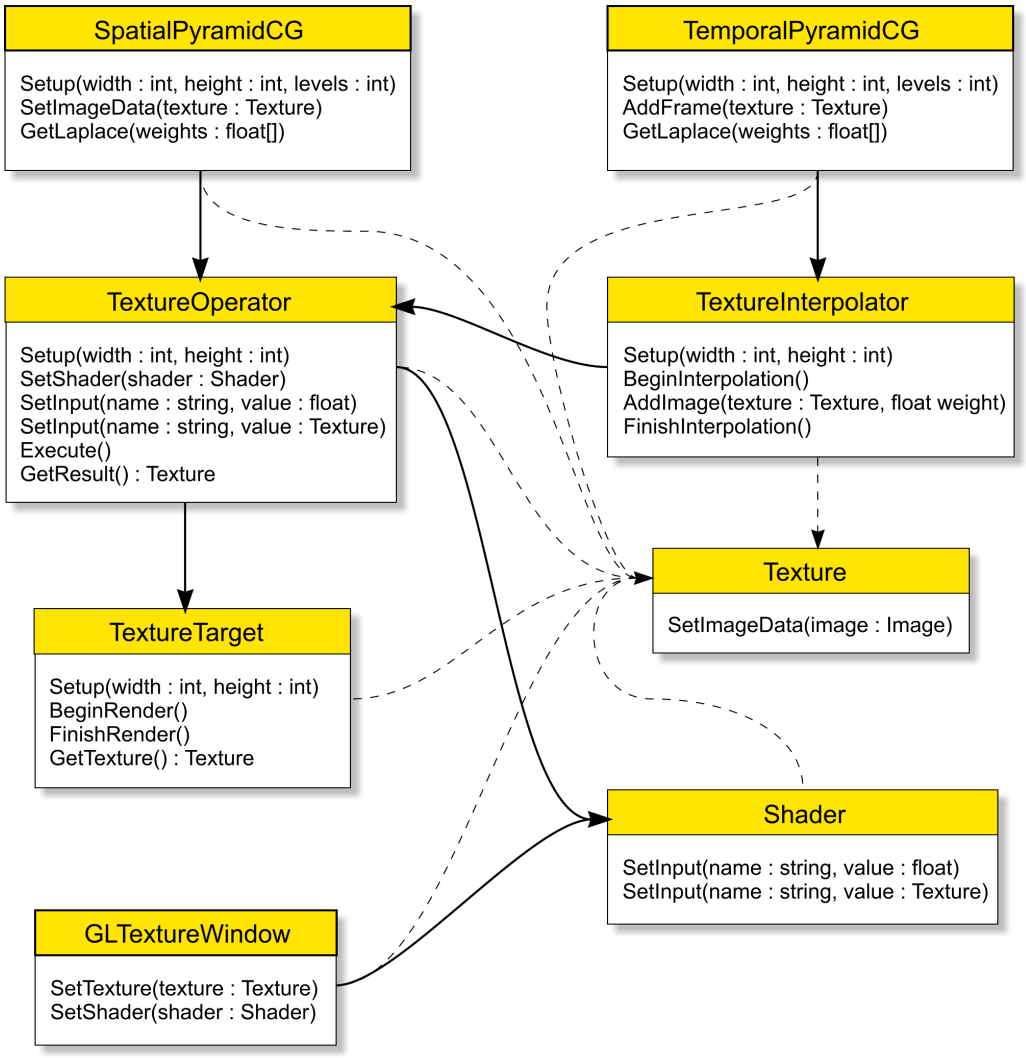


Figure 5: Class structure of the framework

Listing 1: Example application using the framework to filter a video sequence

```
// the video source
VideoSource source

// create the output window
GLTextureWindow window

// setup a temporal pyramid with 5 levels
TemporalPyramidCG tempPyramid
tempPyramid.Setup(source.width, source.height, 5)

// setup a spatial pyramid with 5 levels
SpatialPyramidCG spatPyramid
spatPyramid.Setup(source.width, source.height, 5)

// intermediate textures needed for processing
Texture frameTexture
Texture tempResult
Texture spatResult

// run the main application loop
while source.HasFrames()
  // get the next frame from the video source
  Image frame = source.GetNextFrame()

  // upload the resulting image to a texture
  frameTexture.SetImageData(frame)

  // stream this texture into the temporal pyramid
  tempPyramid.AddFrame(frameTexture)
  // .. and compute a filtered version of the video at the
  // sampling position
  tempResult = tempPyramid.GetLaplace([0.2, 0.5, 1, 1, 1])

  // now filter the temporally filtered result spatially
  spatPyramid.SetImageData(tempResult)
  spatResult = spatPyramid.GetLaplace([0.2, 0.5, 1, 2, 1])

  // display the result
  window.SetTexture(spatResult)
end
```

4.2 The spatial pyramid

4.2.1 Generation

The spatial Laplacian pyramid consists of a specifiable fixed number of N levels, where level 0 has the full resolution image and for each subsequent level the resolution is divided by a factor of two. Each of the levels contains two TextureOperator objects. The first texture operator, 'downsampler', is used to blur and downsample the level for building the next gaussian level. The second one, 'laplaceGenerate', is used for the upsampling and subtraction of a lower level to build the corresponding Lapalace level. See Figure 6 for an overview.

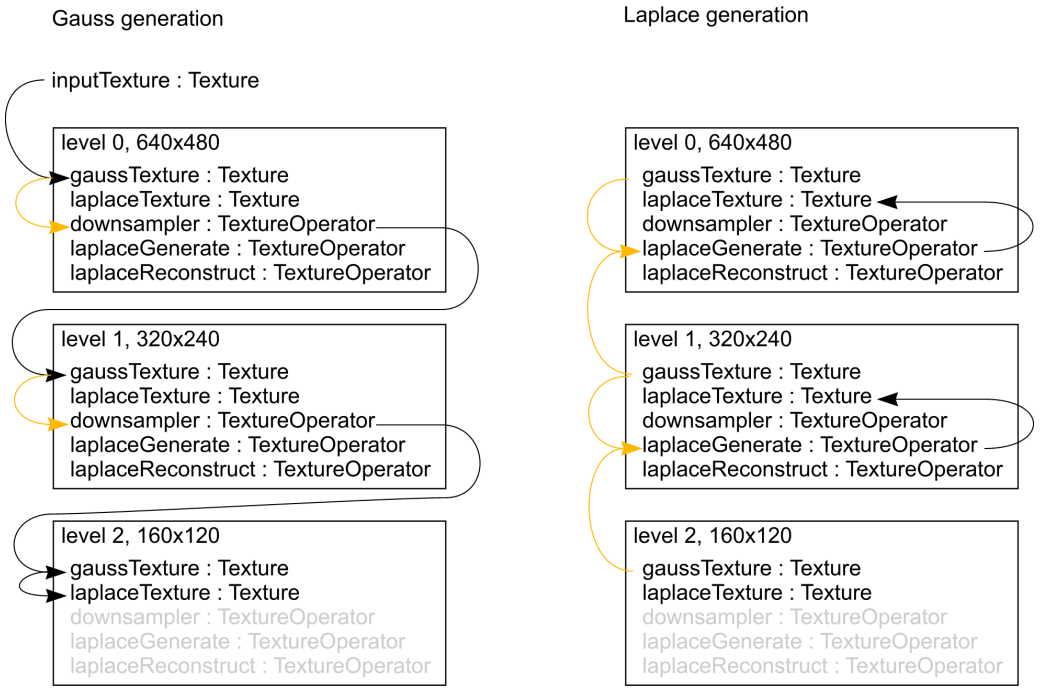


Figure 6: Dataflow for the spatial pyramid construction
The left side shows the generation of the Gaussian part of the pyramid. Then, on the right side, the result is processed in the opposite direction to build the Laplacian images.

The process for building the Laplacian pyramid is now quite simple. First, the input image is downsampled $N - 1$ times to build the downsampled versions for the levels 1 to $N - 1$. Then, the pyramid is processed in the opposite direction, starting from level $N - 2$, up to level 0, where in each iteration two levels are subtracted to get the Laplace image for that level.

Listing 2: Generation of the spatial pyramid

```
method SpatialPyramidCG.SetImageData(inputTexture)
  // level 0 takes the input picture without any processing
  level[0].gaussTexture = inputTexture

  // build the gaussian pyramid...
  for i = 0 to N-2
    level[i].downsampler
      .SetInput("input", levels[i].gaussTexture)
      .Execute()

    level[i+1].gaussTexture
      = level[i].downsampler.GetResult()
  end

  // the lowest resolution laplace level is the same as the
  // lowest gaussian level
  level[N-1].laplaceTexture = level[N-1].gaussTexture

  // build the laplacian pyramid
  for i = N-2 to 0
    levels[i].laplaceGenerate
      .SetInput("low", levels[i+1].gaussTexture)
      .SetInput("high", levels[i].gaussTexture)
      .Execute()

    levels[i].laplaceTexture
      = levels[i].laplaceGenerate.GetResult()
  end
end
```

The downsampling/filtering of the levels is done by the following CG shader:

Listing 3: Downsampling of an image in the Gaussian pyramid

```
#include "filter.cg"

struct foutput {
  float4 color : COLOR;
};

foutput main(float2 texCoord : TEXCOORD0,
             uniform sampler2D input : TEX0,
             uniform float2 delta_input)
{
  foutput OUT;
  OUT.color = gauss5x5l(input, texCoord, delta_input);
  return OUT;
}
```

Listing 4: A 5x5 binomial filter exploiting linear texture sampling

```
// in filter.cg
// ...
const float gauss_kernel_4x4[4][4] = {
    {0.015625, 0.046875, 0.046875, 0.015625},
    {0.046875, 0.140625, 0.140625, 0.046875},
    {0.046875, 0.140625, 0.140625, 0.046875},
    {0.015625, 0.046875, 0.046875, 0.015625},
  };
// ...
float4 gauss5x5l( sampler2D tex, float2 tc, float2 delta ){
  tc -= 0.5*delta;
  float4 c = 0;
  for( int y = -1; y <= 2; y++ ){
    for( int x = -1; x <= 2; x++ ){
      c += gauss_kernel_4x4[x+1][y+1]
          * tex2D(tex, tc + float2(x,y)*delta);
    }
  }
  return c;
}
// ...
```

This version of the 5x5 binomial filter is exploiting the fact, that the input

texture is automatically sampled with linear interpolation by the GPU. The source texture is sampled halfway between the texel centers to yield the average value of the four neighbouring texels, which corresponds to a 2x2 binomial filter. These values are then filtered through a 4x4 binomal filter, yielding the final 5x5 filtered result.

For the generation of the corresponding Laplace image, the following shader is used:

Listing 5: Generation of a Laplace level given two Gaussian levels

```
#include "filter.cg"

struct foutput {
  float4 color : COLOR;
};

foutput main(float2 texCoord : TEXCOORD0,
             uniform sampler2D low : TEX0,
             uniform float2 delta_low,
             uniform sampler2D high : TEX1,
             uniform float2 delta_high)
{
  foutput OUT;

  // upsample the low resolution level and subtract from
  // the high resolution one
  OUT.color = tex2D(high, texCoord) -
      upsample_2_gauss5x5l(low, texCoord, delta_low);

  // add a bias to account for the unsigned texture format,
  // which can only store values from 0 to 1
  OUT.color = OUT.color*0.5 + 0.5;

  return OUT;
}
```

Listing 6: Upsampling of an image by a factor of two

```
// performs the equivalent of inserting a black pixel
// between every two neighbouring pixels in the source
// texture and then applying a gaussian 5x5 filter.
// This version exploits the linear texture sampling,
// like the downsampling filter does.
float4 upsample_2_gauss5x5l(
    sampler2D tex,  // the source texture
    float2 tc,      // texture coordinate [0 .. 1]
    float2 delta    // tex. coord. delta of one texel
  )
{
  float2 texel = tc/delta;
  float2 texel_l = floor(texel);
  bool2 center = texel-texel_l < 0.5;
  const float2 toff = 0.25;
  if( center.x ){
    if( center.y ){
      return 0.25 * (
         tex2D(tex, tc + (float2(-0.2, -0.2)+toff)*delta) +
         tex2D(tex, tc + (float2(+0.2, -0.2)+toff)*delta) +
         tex2D(tex, tc + (float2(-0.2, +0.2)+toff)*delta) +
         tex2D(tex, tc + (float2(+0.2, +0.2)+toff)*delta)
        );
    } else {
      return 0.5 * (
         tex2D(tex, tc + (float2(-0.25, 0)+toff)*delta) +
         tex2D(tex, tc + (float2(+0.25, 0)+toff)*delta)
        );
    }
  } else {
    if( center.y ){
      return 0.5 * (
         tex2D(tex, tc + (float2(0, -0.25)+toff)*delta) +
         tex2D(tex, tc + (float2(0, +0.25)+toff)*delta)
        );
    } else {
      return tex2D(tex, tc+(float2(0,0)+toff)*delta);
    }
  }
}
```

4.2.2 Reconstruction

For the reconstruction of a modified output image, the pyramid is processed again from level $N-1$ to level 0. Each level is multiplied by its corresponding weight and then added to the upsampled result of the previous level, if any. This is the same algorithm as outlined in Figure 1.

Listing 7: Reconstruction of the modified spatial image

```
method SpatialPyramidCG.GetLaplace(weights[N])
  // begin with the lowest level
  result = level[N-1].laplaceTexture

  // iterate up to the highest level
  for i = N-2 to 0
    if i == N-2 then
      lowWeight = weights[N-1]
    else
      lowWeight = 1
    end

    level[i].laplaceReconstruct
      .SetInput("low", result)
      .SetInput("lowWeight", lowWeight)
      .SetInput("high", level[i].laplaceTexture)
      .SetInput("highWeight", weights[i])
      .Execute()

    result = level[i].laplaceReconstruct.GetResult()
  end

  return result
end
```

Listing 8: Spatial reconstruction shader

```
#include "filter.cg"

struct foutput {
  float4 color : COLOR;
};

foutput main(float2 texCoord : TEXCOORD0,
             uniform float lowWeight,
             uniform sampler2D low : TEX0,
             uniform float2 delta_low,
             uniform float highWeight,
             uniform sampler2D high : TEX1,
             uniform float2 delta_high)
{
  foutput OUT;

  // upsample the low texture by a factor of two,
  // using a 5x5 gauss filter
  OUT.color = lowWeight *
      upsample_2_gauss5x5l(low, texCoord, delta_low);

  // add the high resolution texture, removing the bias
  OUT.color += highWeight *
      (tex2D(high, texCoord)*2.0 - 1.0);

  return OUT;
}
```

4.3 The temporal pyramid

4.3.1 Generation

The temporal pyramid is very simple on the GPU side, as only weighted sums of full images have to be computed and no image space filtering is happening. Conceptually, for a pyramid of M levels, the images are stored in a cyclic array of size 2^{M+1}. Each consecutive level has half the number of frames, each computed by applying the 5-tap binomial kernel to its neighbouring frames in the previous level. The actual sampling using GetLaplace() is done at the index $2^{M+1} - 2$. This is required so that the generation and reconstruction processes, which conceptually run in parallel for the lifetime of the pyramid, do not overlap in the cyclic array. See Figure 7 for a visualization of these areas. The latency caused by this requirement makes the temporal filtering suitable only for prerecorded videos, where video latency is not an issue.

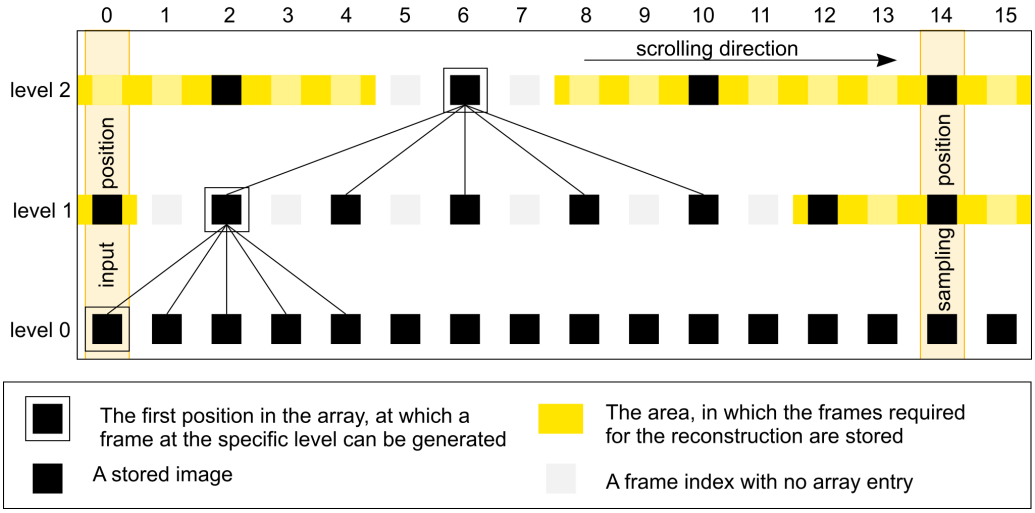


Figure 7: The structure of the temporal pyramid

In this case, only the Gaussian pyramid is stored in memory. The Laplacian images will be computed on the fly to reduce the used amount of memory bandwith (assuming only one output image will be computed per frame).

Before the pyramid update algorithm can run, a filter kernel has to be computed for each level, which is used for proper upsampling any level to level 0 (highest resolution). This is done in the Setup() method:

Listing 9: Initialization of the temporal pyramid

```
// M is the number of levels in the pyramid
method TemporalPyramidCG.Setup(width, height, M)
  // compute the number of samples/images
  // in the highest level
  W = 2^(M+1)

  // set the frame counter to zero
  currentFrame = 0

  // setup all levels of the pyramid
  for l = 0 to M-1
    // the number of samples in this level
    W[l] = W / 2^l

    // setup all samples of this level
    for j = 0 to W[l]-1
      level[i].sample[j].baseIndex = j
      level[i].sample[j].downsampler
        .Setup(width, height)
        .SetShader("temporalpyramid_downsample")
    end

    // compute the upsampling kernel for this level
    if l == 0 then
      level[l].kernel = [1]
    else
      level[l].kernel
        = convSpread(level[l-1].kernel, gauss_kernel_5)

      // convSpread(A, B):
      //   A' = insert a 0 after each sample of A
      //   return the convolution of A' and B
    end
  end

  // set the reconstruction position
  samplingPosition = W - 2

  // setup the interpolator
  interpolator.Setup(width, height)
end
```

The AddFrame() method then scrolls the cyclic array by one position and inserts the new image into the first array position of the highest pyramid level. Then, for each successive level, the first possible position to compute a new sample from its five children is checked for an array entry. If an entry exists at that position, the new sample is computed and stored there. See Figure 7 for a visualization of these positions.

Listing 10: Algorithm for appending a sample to the temporal pyramid

```
method TemporalPyramid.AddFrame(texture)
  // advance the frame counter
  currentFrame = currentFrame + 1
  // and update the first sample
  level[0].sample[currentFrame mod W[0]].image = texture

  // check in each level whether there is an image that can
  // be computed now
  // firstPossiblePos is the first position for each level,
  // where a sample of that level can be computed using its
  // 5 high resolution child samples in the previous level.
  firstPossiblePos = 2
  for l = 1 to M-1
    if ImageExists(l, firstPossiblePos) then
      ComputeGaussImage(l, firstPossiblePos)
    end
    firstPossiblePos = firstPossiblePos + 2*2^l
  end
end

method TemporalPyramid.ComputeGaussImage(l, frame)
  for i = -2 to 2
    cidx = GetChildIndex(l, frame, i)
    level[l].downsampler
      .SetInput("input[i+2]", level[l-1].image[cidx])
  end

  level[i].downsampler.Execute()
  level[i].image = level[i].downsampler.GetResult()
end

method TemporalPyramidCG.ImageExists(level, frame)
  return GetGaussImageIndex(level, frame) != -1
end
```

```
method TemporalPyramidCG.GetChildIndex(level, frame, offset)
  // compute the position of the specified child
  childframe = frame + offset*2^(level-1))

  // return the index of the image at that position
  return GetGaussImage(level-1, childframe)
end

method TemporalPyramidCG.GetGaussImageIndex(level, frame)
  if frame < 0 or frame > currentFrame then
    return -1
  end

  // index relative to the current frame
  offset = currentFrame - frame

  // corresponding sample in the specified level
  offset_level = offset / 2^level

  // check if we are actually between two frames
  if offset_level * 2^level != offset then
    return -1
  end

  // we hit a frame exactly, return
  // the actual index into the level array
  return idx_level mod W[level]
end
```

4.3.2 Reconstruction

The reconstruction of a final output image (i.e. a weighted sum of the Laplacian basis images) is done by upsampling all Gaussian levels up to full resolution and computing a weighted sum of those – the pairwise subtraction is done implicitly by modifying the weights accordingly. The upsampling of the low resolution levels $N-1$ to 1, in this case, is not done by iteratively doing an upsampling of factor 2. Instead, the full level of resolution is computed at once for each level. This saves a large amount of memory and memory bandwith, which would otherwise be required for the intermediate frames. Also, this approach is computationally (number of texture accesses, additions, and multiplications) cheaper, as long as it is done only once per

frame. See Figure 8 and 9 for a rough comparison of the two methods. The reconstruction algorithm now looks as follows:

Listing 11: Reconstruction algorithm for the temporal pyramid

```
method TemporalPyramidCG.GetLaplace(weights[M])
  interpolator.BeginInterpolation()

  for i = 0 to M-1
    if i < M-1 then
      // subtract two (gauss) levels to get the
      // corresponding laplace level
      UpsampleLevel(i, weights[i])
      UpsampleLevel(i+1, -weights[i])
    else
      // the lowest gauss level is equivalent
      // to the lowest laplace level
      UpsampleLevel(i, weights[i])
    end
  end

  return interpolator.FinishInterpolation()
end

method TemporalPyramidCG.UpsampleLevel(l, weight)
  kernelSize = level[l].kernel.size
  kernelOffset = -kernelSize / 2

  // convolve the upsampling filter with the level array
  for i = 0 to kernelSize-1
    frame = samplingPosition + kernelCenter + i
    idx = GetGaussImage(l, frame)

    // if we hit a frame here,
    if idx != -1 then
      // add it with the according weight
      imageweight = weight * level[l].kernel[i]
      interpolator.
        AddImage(level[l].image[idx], imageweight)
    end
  end
end
```

5 Optimization

In the various parts of the system, several optimizations to the naïve algorithm have been employed. However, there is still room for further optimization. Some of these points will be listed in this section.

5.1 Spatial filtering

For spatial filtering, the texture accesses have been reduced by exploiting the fact that sampling in the center of four neighbouring texels is equivalent to performing a 2x2 binomial kernel convolution of the texture. This is a side-effect of the linear interpolation, which the hardware provides for texture sampling. The number of texture accesses for performing a 5x5 kernel convolution are therefore reduced from 25 to 16. The upsampling filter goes a bit further by adjusting the texture coordinates to different fractions to get the desired weights.

Further reduction can be achieved by filtering the images seperately in the two spatial dimensions. This is possible because the used binomial kernel is a separable filter and thus applying a 1x5 binomial kernel followed by a 5x1 one gives the same result as applying a 5x5 kernel directly. This further reduces the number of per-pixel accesses to 10 or 8, respectively, at the cost of some memory and additional write accesses.

Currently the slowest part of the spatial pyramid is the upsampling filter. The shader (Listing 6) contains a large number of alternative execution paths with different texture sampling operations. This has a severe impact on performance – especially on cards which do not support native conditional code execution. A precomputed upsampling texture could be used to simplify the shader to five texture accesses without the conditional logic.

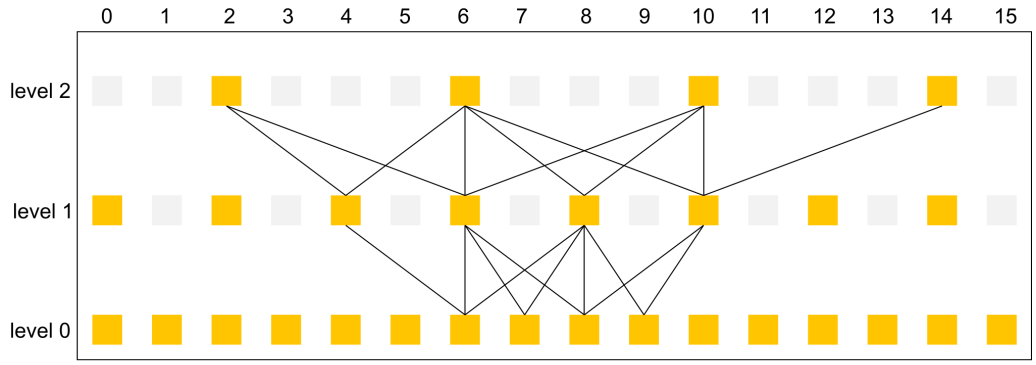
5.2 Temporal filtering

For temporal filtering, the main performance issue is the huge memory requirement. A full Laplacian pyramid of M levels would require roughly 2^{M+3} times the size of a single video frame. For a HD video with a resolution of 1280x720 this amounts to almost 700MB of data for 5 pyramid levels (corresponds to a kernel size of about one second at 30 Hz), which pushes even the highest end graphic cards to their limits.

The approach in this implementation is to store only the Gaussian pyra-

mid. That reduces the number of stored frames to about 2^{M+2}, which leads to about 340MB memory required for the previous example. A single reconstruction will also require about half of the number of texture accesses compared to the basic algorithm. This is sketched in Figures 8 and 9.

For a further reduction of the memory use by a factor of two, the video frames could be handled as three seperate textures, where the U and V components would have half the resolution of the Y texture. This approach would however add considerably to the implementation complexity. An alternative would be to use an appropriate OpenGL extension (EXT_422_pixels, APPLE_ycbcr_422, MESA_ycbcr_texture). Unfortunately, those extensions don't seem to have widespread support on mainstream OpenGL implementations.



Number of texture-sampling operations required on average for reconstructing four successive frames

1. Samples required for building the gaussian samples: 3*5 =	15
2. Samples required for building the laplacian samples: 5+10 =	15
3. Samples required for reconstructing level 2 to 1: 5+2 =	7
Samples required for reconstructing level 1 to 0: 10+4 =	14
Samples required for sampling level 0:	0
Samples required for complete reconstruction: 15+15+7+14 =	**51**

Figure 8: Successive reconstruction of three temporal levels

5.3 Concurrency

When using the GPU to reduce the workload of the CPU, it is important to schedule the operations in a way that maximizes concurrency. The only considerable task of the CPU in this case is to decode the video frame and send that to the GPU. To improve the concurrency, this process is moved from the beginning of the frame processing to the middle. That way, while the GPU is processing the current video frame, the CPU is already busy decoding the next one.

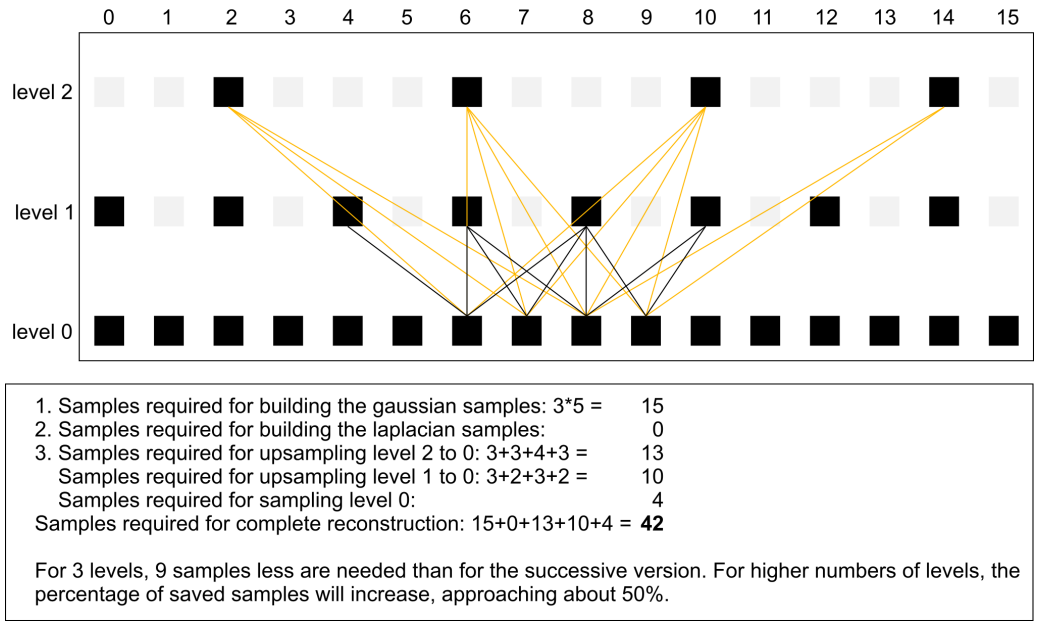


Figure 9: Direct reconstruction of three temporal levels

6 Results

Some comparison benchmarks were done to compare the performance of the CPU based algorithm to the new GPU accelerated one. The video used was a HDV video with a resolution of 1280x720 pixels. The reconstruction was done with uniform weights.

Figure 10 shows the results of a benchmark run on a notebook for different numbers of temporal and spatial levels. The low performance notebook graphics chip, a GeForce Go 7600, is the limiting factor in this case and causes the performance degradation for the spatial filtering. Temporal filtering, which only uses light-weight shaders, exhibits a relatively stable performance until the amount of memory required to store the pyramid images exceeds the graphics memory (256 MB) with 5 levels.

In Figure 11, the results of a desktop system with a fast GeForce 8800 GT are shown. The limiting factor in this case is memory transfer bandwith, which limits the framerate to about 36 fps, even without any filtering. Again, for temporal filtering, the performance starts to drop when the graphics card memory (512 MB) is filled up.

The system in Figure 11 performs much better than the system in Figure 12, albeit having a much slower GPU. This again shows the heavy dependence on memory transfer bandwith. The performance breakdown at five temporal levels is clearly visible in this case.

All tested systems, except the notebook system with a particulary slow GPU, were memory transfer bandwith limited. This was the case with the GPU algorithm, as well as the CPU version. Note that the CPU version does not exploit multiple cores. However, because of the memory dependence, it is expected to perform similar with a multithreaded algorithm.

An average performance gain factor of about 2.1 and 3.7 was measured for four levels of spatial and temporal filtering, respectively. The GPU could be successfully employed to perform full spatio-temporal Laplacian filtering of high resolution videos in realtime.

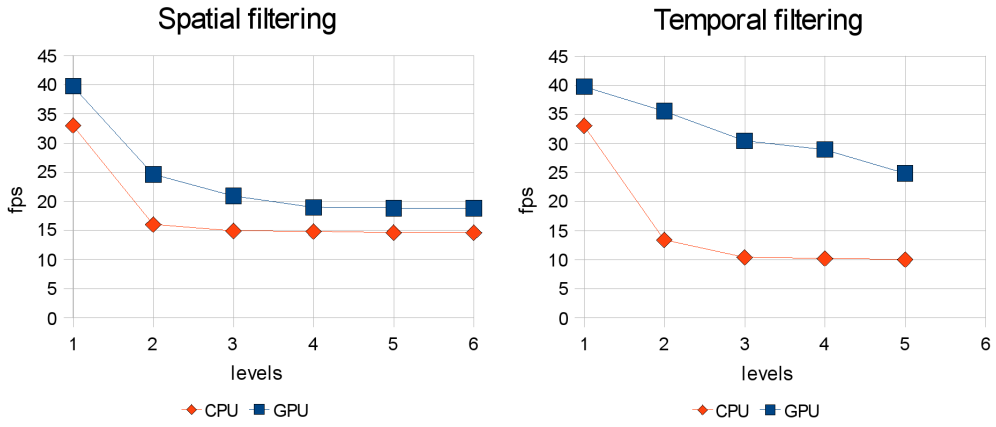


Figure 10: Performance on an Intel Core Duo (2x1.66 GHz), 1GB RAM, Nvidia GeForce Go 7600 with 256 MB RAM; 1280x720 MPEG2

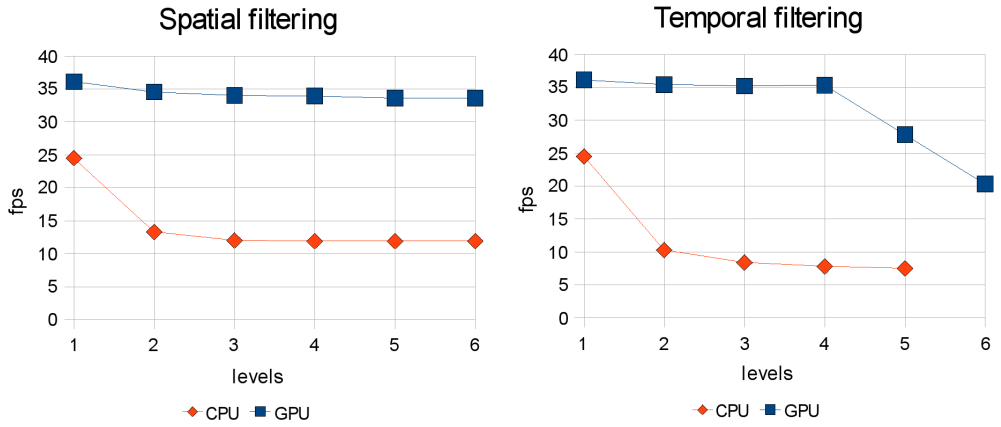


Figure 11: Performance on an AMD Athlon 64 3700+, 1 GB RAM, Nvidia GeForce 8800 GT with 512 MB RAM; 1280x720 MPEG2

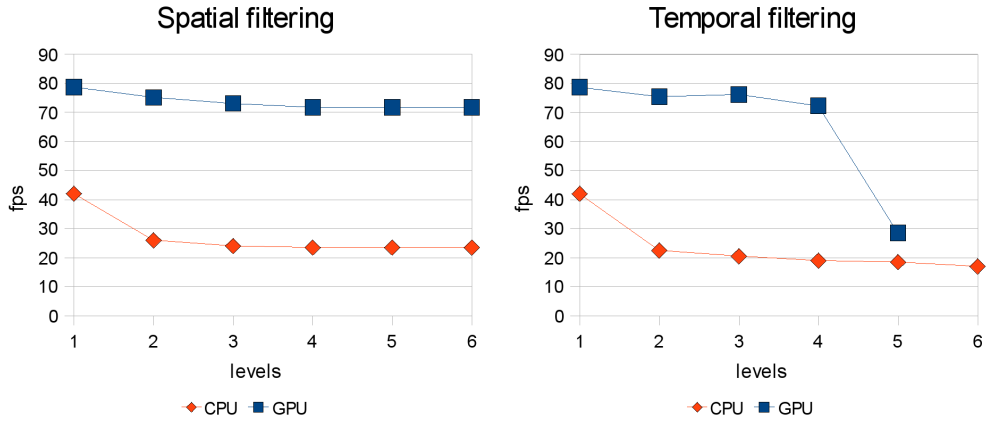


Figure 12: Performance on an Intel Core 2 Quad, 2 GB RAM, Nvidia GeForce 8600 GT with 512 MB RAM; 1280x720 MPEG2

List of Figures

Listings

References

[1] Edward H. Adelson and Peter J. Burt. Image data compression with the laplacian pyramid. In *PRIP81*, pages 218–223, 1981.

[2] Martin Böhme, Michael Dorr, Thomas Martinetz, and Erhardt Barth. A temporal multiresolution pyramid for gaze-contingent manipulation of natural video. In Riad I. Hammoud, editor, *Passive Eye Monitoring*, chapter 10. Springer, 2007. (in print).

[3] Giorgio Bonmassar and Eric L. Schwartz. Improved cross correlation operator for template matching on the Laplacian pyramid. *Pattern Recognition Letters*, 19(8):765–70, 1998.

[4] Peter J. Burt and Edward H. Adelson. The laplacian pyramid as a compact image code. *IEEE Transactions on Communications*, COM-31,4:532–540, 1983.

[5] Peter J. Burt and Edward H. Adelson. A multiresolution spline with application to image mosaics. *ACM Trans. Graph.*, 2(4):217–236, 1983.

[6] Michael Dorr. Effects of Gaze-Contingent Stimuli on Eye Movements. Diploma thesis, Universität zu Lübeck, 2004.

[7] Andrew T. Duchowski. Hardware-accelerated real-time simulation of arbitrary visual fields. In *ETRA '04: Proceedings of the 2004 symposium on Eye tracking research & applications*, pages 59–59, New York, NY, USA, 2004. ACM.

[8] Wilson S. Geisler Jeffrey S. Perry. Gaze-contingent real-time simulation of arbitrary visual fields. In *Human Vision and Electronic Imaging, SPIE Proceedings*, 2002.

[9] Shuguang Mao and Morgan Brown. Pyramid-based image synthesis - theory. *Stanford Exploration Project*, 2002.