Flexible Film: Interactive Cubist-style Rendering

M. Spindler† and N. Röber and A. Malyszczyk and T. Strothotte

Department of Simulation and Graphics
School of Computing Science
Otto-von-Guericke University Magdeburg, Germany

Abstract

This work describes a new approach of rendering multiple perspective images inspired by cubist art. Our implementation uses a novel technique which not only allows an easy integration in existing frameworks, but also to generate these images at interactive rates. We further describe a cubist-style camera model that is capable of embedding additional information, which is derived from the scene. Distant objects can be put in relation to express their interconnections and possible dependencies. This requires an intelligent camera model, which is one of our main motivations. Possible applications are manifold and range from scientific visualization to storytelling and computer games.

Our implementation utilizes cubemaps and a NURBS based camera surface to compute the final image. All processing is accomplished in realtime on the GPU using fragment shaders. We demonstrate the possibilities of this new approach using artificial renditions as well as real photographs. The work presented is work in progress and currently under development. To illustrate the usability of the method we suggest several application domains, including filming, for which this technique offers new ways of expression and camera work.

1. Introduction

The familiar representation of objects in our environment is that they usually face us with one side only, except they are viewed from odd angles or mirrored in reflective surfaces. An often expressed desire of artists and scientist throughout the centuries was the combination of different viewpoints of objects and scenes into a single image. Cubism was one of the biggest art movements that explicitly focussed on these characteristics. Cubism was developed between about 1908 and 1912, and a collaboration between Pablo Picasso and Georges Braque. The cubistic movement itself was short and not widespread, but it started to ignite a creative explosion that resonated through all of 20th century art and science. The key concept of Cubism is that the essence of objects can only be captured by showing it from multiple points of view simultaneously, thus resulting in pictures with multiple perspectives [9]. Other examples can be found in ancient panoramic drawings of China and the work from M.C. Escher and Salvadore Dali.

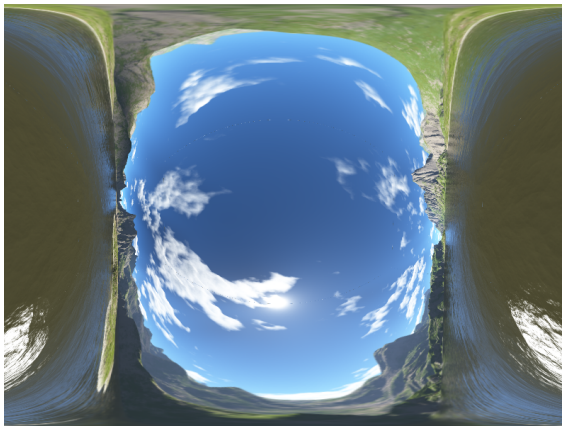

Figure 1: *Super Fisheye View. (Cubemap from [29])*

Fascinated by these ideas and possibilities, many artists and scientists picked up the concept of multiple perspectives, extended it or adopted it to new areas. With the technological advances and the increased flood of images in the recent

† spindler@isg.cs.uni-magdeburg.de

decades, *new* ways of looking at objects are more important than ever. Cubism might be able to provide a solution to this problem by combining the information of several images into one. Although cubistic images challenge the eye and the mind by its unnatural representation of objects, yet it can be more efficient in the visualization of certain actions and events. Several lines of research have evolved that focus on the generation and the useful utilization of cubist-style imagery, including art, science and especially computer graphics.

One of our main motivations for the development of a cubist-style camera model was the desire of an intelligent camera system, which is able to select the most important views of a scene and combines them into a single image or animation. This key feature of Cubism has already been applied to many applications, including comics and computer games. Ubisoft extended an existing game engine by additional comic elements, like insets or onomatopoeia, which not only amplify the comic like character of the game, but also add a multi-perspective component to certain scenes [30]. As Ubisoft used hard cuts between scenes only, we are interested in camera work and composition techniques that allow the gradual transition of a single into a multiple perspective representation of the environment. Along this research, we are also interested in the narrative possibilities for conveying additional scene and story relevant information. These techniques can also be used for film animations, to develop new styles of cubistic movies [28].

Many research articles have affected our work, but the ones that influenced our research most are the publications by Wyvill [33], Rademacher [27] and especially the technical report by Glassner [14]. Wyvill et.al. [33] probably published the first research article in computer graphics concerning alternative, non-planar camera models. Later Rademacher [27] employed curved camera surfaces by moving a slit camera along animated pathways. Glassner [14], and later [12], [13], picked up the same idea and developed a plugin for 3D Studio MAX to render cubistic scenes. What we found most impressive in Glassner's first article [14] was not the implemented technique nor the achieved results, but the hand-drawn figures which he used to visualize the idea and the possibilities of cubist-style rendering, see also Section 2.3. So far we have not found any technique capable of rendering such images, but we believe that it might be possible using our approach.

As we toyed with the idea of integrating our camera model into a 3D game engine, to explore the possibilities regarding the story telling and the game play, we had to find ways for the interactive rendering of such multi-perspective images. Our system uses environment mapping and a flexible camera surface for the rendering. The scene is rendered through six cameras, which are used to compile a cubic environment map of the scene. At the moment we experiment with static cameras only, but a realtime update of the cubemap is possible. The camera surface is described by several NURBS functions, which model a smooth and flexible film surface. This surface is sampled using fragment shaders on the GPU, and the final image displayed on the screen. We have developed several implementations, including some optimization to reduce the number of computations.

The paper is organized as follows: The next section gives an exhaustive overview of related work and discusses non-planar camera models with single and multiple perspectives. We start with a retrospective of the classic, single perspective camera model which leads over to distortion techniques like fisheye lenses, and finally converges in the discussion of multiple perspective rendering algorithms. The section is closed with remarks towards the rendering of cubist-style images. The following two sections explain our approach en detail, with Section 3 focussing on the theoretical aspects and Section 4 highlighting the implementation details. Section 5 presents results and shows images that were generated using our technique. In addition, we show performance results and discuss optimization issues. Finally, in Section 6 we conclude the paper and summarizes the work and point out areas for future improvements.

2. Non-planar Camera Projections

Most computer imagery is created using classic, single perspective camera models. The first camera developed was the camera obscura, a simple pinhole camera which is known to artists and scientist since antiquity [9]. Pinhole cameras only use (a very small) aperture and create a fully reversed picture on the opposite side of the hole. This simple model was further extended over the centuries and finally used by the Lumière brothers in the late 19th century to create the first moving pictures [4], [9].

From a computers perspective, the classic camera model is defined by the cameras position and orientation, as well as the cameras aperture angles. Another interesting feature are the number of vanishing points: points at which parallel lines run off into the horizon. As these points define the cameras perspective, a different number of vanishing points are often used depending on the application. Examples can be found in CAD systems and computer games, where perspectives with varying numbers of vanishing points are used, often depending on the game genre [21].

Besides the perspective camera model, scientist, and especially artists, have worked on alternative representations for object and entire scenes. Da Vinci, Dali, Escher and Picasso are the most prominent who worked on different views and variant camera models. With the advances of computer graphics, this topic moved into the focus of several researchers working in computer science. Wyvill first discussed alternative camera models in computer graphics by using a flexible camera surface and a raytracing algorithm for the rendering [33]. Reinhart developed a new filming

technique by swapping spatial and temporal axes of the media. Depending on the cameras orientation, it allowed him to visually accelerate or decelerate certain actions [28].

The next sections discusses several existing camera models which employ a non-planar camera surface or an image distortion technique. We first start with a discussion on single perspective camera distortions, which we later extend to multiple perspectives. The end of this section focusses on Cubism and true cubistic rendering techniques.

2.1. Single Perspectives

Several image distortion techniques have been developed that conserve a single point of view, but are able to focus on specific parts of an image for highlighting purposes. These approaches can be divided into object-space and image-space techniques. As the names suggest, object-space techniques distort the underlying model (3D mesh) in order to enhance certain parts, while image-based methods work on the rendered image and eventually employ an additional camera.

Diverse articles on mesh distortion have been published, in which the underlying 3D model is deformed depending on the viewers orientation or for accentuation. This includes the research on view dependent geometry by Rademacher [26], in which key viewpoints are used to deform the mesh according to the current perspective. Martin et.al [20] implemented a similar system to simulate sketchy illustrations. Isenberg [17] developed a system that applies two dimensional distortions in order to create 3-dimensional variations of stroke based models and 3D polygonal meshes. The applications, for which all of these systems were designed for, are sketchy/illustrative rendering of 3D models and mesh animation. Another interesting implementation of mesh distortions are the zooming techniques developed by Raab [25] and Preim [24]. These techniques allow the accentuation of certain parts of a model using a fisheye zoom, as well as the drawing of explosion and implosions diagrams to visualize the construction of the scene.

In addition to object-space distortions, several image based techniques have been developed. Some of them are concerned with the utilization of additional lenses to highlight and magnify certain portions of the images. A classic technique is the use of fisheye lenses to extend the range of classic camera systems [19], [4]. A similar method was presented by Carpendale et.al [5], who applies additional lenses on rendered images to magnify parts of the image. Additional object-space techniques were developed to allow a supplementary focus on particular mesh objects. Another object/image-space technique is the RYAN system with the Boss camera, which is integrated into the Maya rendering package [7]. Although it includes mesh distortions, it generally focusses on the deformation of the projections to render artistic, nonlinear images. Other techniques focus on special camera work and the animation of still pictures. Horry et.al. [15] designed a system that utilizes a spidery mesh and allows a "Tour into a Picture". Based on this principle Chu et.al. [6] extended this technique and developed a multiple perspective camera system for animating Chinese landscape panoramas. Opposed to the previously discussed techniques, Böttger explores the possibilities of combining very small and very large scale scenes together by using a logarithmic approach [3]. Furthermore, he develops a system to visualize the visual examination of objects.

2.2. Multiple Perspectives

While the last section discussed methods using single perspectives only, this section focusses on composing several viewpoints together into one image. Although the underlying principle is the same, these methods can be used for two kinds of multi-perspective rendering: object and scene cubism. The first shows an object from different angles and allows the perception of several sides of this object simultaneously. The other technique is used to combine several objects of a scene and to visually re-arrange them. Regarding a flexible camera surface that captures the environment, the first technique uses a more concave surface, while the second utilizes a more convex shaped area. To exemplify this idea, the camera surface of a true fisheye lens would have a spherical appearance.

A simple technique that allows an easy rendering of multi-projection images is the slit camera system, that was discussed by Glassner [14] and is also used in many other implementations [27], [34]. As the name suggests, a slit camera only exposes parts of the film (a slit) at a time by simultaneously moving the camera along an animated path. Glassner implemented a multi-perspective camera system using two NURBS surfaces (eye and lens) as render plugin for 3D Studio MAX, and used the internal raytracer for the image generation [14]. Wood et.al. describe a method for simulating apparent camera movements through 3D environments [32] by employing a multi-pinhole camera system and image registration. The technique is used to render multi-perspective panoramas and motivated with the utilization for cel animations. Rademacher discusses Multiple-Center-of-Projection images and develops a rendering technique by using a slit-camera approach and image registration [27]. The proposed method works equally well for real-world, as well as for artificial data sets. A similar system for the direct rendering of multi-perspective images was proposed by Yu at.al. which includes a sampling correction scheme to minimize distortions [34].

In addition, Agrawala et.al. [1] discuss a rendering system for multi-perspective rendering with the focus on artistic image generation. They designed a tool that uses local cameras and allows the re-orientation of each object individually in the final composition, thus rendering multi-projection images. A raytracing method of rendering multi-perspective

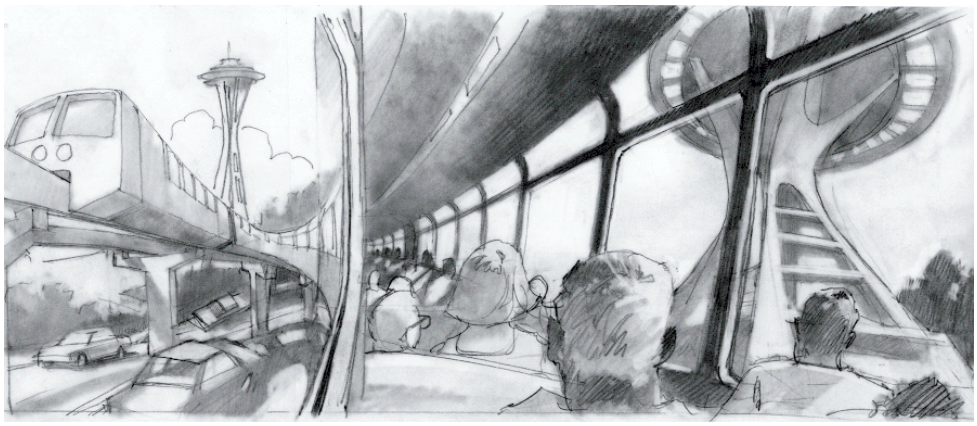

Figure 2: *The Seattle Monorail as seen from a cubistic viewpoint. (from [14])*

images was proposed by Vallance et.al. [31]. He provides an extensive overview of multi-perspective image generation and proposes an OpenGL like API for rendering such images using a flexible Bézier camera surface and raytracing.

2.3. Cubist-style

The essence of Cubism is often referred to the presentation of objects from multiple angles, therefore composing several points of view into one single image. These view points are combined as facets of different perspectives and act as a grid like structure of the image. The term Cubism was first used by the French art critic Louis Vauxcelles, as he described the work as "bizarre cubiques". Subsequently, a name for this art movement was defined [9].

The majority of research articles discussed so far was only concerned with the first point of cubism: the rendering of images with multiple centers of projection. Only very few are true Cubist-style rendering techniques, including the work by Collomosse et.al [8] and Klein at.al. [18]. The system by Collomosse uses several 2D images, showing an object from different viewpoints, and image registration of salient features to compose the final result. This composition is additionally rendered using a painterly effect to closely mimic the effects of true cubist art. A different approach is described by Klein et.al. [18] who utilizes a video cube, similar to Reinhart [28], to render multi-perspective images. Similar to the work of Collomosse, the resulting compositions are further modified trough stylistic NPR drawing techniques, including painterly effects and mosaics.

Besides the rendering of true cubist art, the most intriguing feature seems to be the composition of several viewpoints into one image. This can be easily derived from the number of articles focusing on this topic, but also on the applications that were developed for it. From all research articles we have seen, we found the hand-drawn images of Glassner's paper, see Figure 2, to be the most impressive [14]. Although their are not computer generated, we see them as our goal for the rendering of multiple perspective imagery. A characteristic that these images posses and all others lack is the natural, organic look through non-photorealism. A direction for future research therefore should be the integration of additional NPR drawing styles and the generation of non-perfect images. Non-photorealistic rendering has established itself as an indecent area of research and successfully applied to many areas [11], [10].

In our implementation we are currently focussing on the first aspect of Cubism only: the rendering of multi-perspective images, but our framework already allows the additional input of NPR techniques. At the moment we find it more challenging to work with the camera model itself, and to research the possibilities for narration and storytelling.

3. Flexible Film

In this section we explain and discuss our Flexible Film camera model and the related techniques in algorithmic depth. The following Section 4, comes back to the here discussed methods and presents implementation details and code fragments.

As illustrated earlier in the introductory part, our goal is to create an intelligent camera system, which can be integrated into 3D environments, like game engines, and which support the user by generating meaningful images that enhance the depicted scene. Our current work describes the rendering part of this system, which is able to produce multi-projection images in realtime. At the moment, this method only works for static cameras, i.e. cameras that stay at a fixed location, but we have already explored the possibilities for the extension towards dynamic scenes. A difficulty in multi-

perspective camera systems is the control of the camera itself, especially for camera movements and animations. Here we have developed a very simple, yet effective technique, which will evolve into an API that can be easily integrated in existing applications.

Our method is a mixture between image based and direct rendering. The environment is represented through a cubemap, which is compiled prior the actual rendering process. Thus, our method can be described as image based rendering. In addition, the cubemap can also be updated during the rendering process, which allows the representation of dynamic scenes and camera movements, hence our method is similar to direct rendering. A flexible camera surface, which we call *FlexibleFilm*, is placed within and used to lookup the cubemap, depending on the actual shape of the mesh. This surface is described by NURBS functions and implemented into a Cg fragment shader. This allows an easy and interactive rendering of multiple perspective images.

The next two sections describe our Flexible Film system in algorithmic depth. While the first section focusses on the surface sampling and the necessary techniques, the second one describes our camera model, which sits on top of the surface and is used to control the flexible camera surface.

3.1. Surface Sampling

The heart of our rendering system is the flexible camera surface that is used to sample the cubic environment in order to determine the final composition. This surface is represented by a NURBS function and sampled each rendering cycle using a Cg fragment shader and a customized fragment stage.

We proceed as follows: After placing an orthographic camera at position $(0,0,1)$ looking into the negative z direction, we draw a quad at $z = 0$ that entirely fills the screen. Then, we assign texture coordinates to its four corners ranging from $(0,0)$ to $(1,1)$, where the former one is the lower left corner and the latter the upper right corner. This allows us to arithmetically walk through a normalized NURBS surface, which is typically defined in this specific domain. By abusing fragment shader, we plug the incoming interpolated texture coordinates (u,v) into the Bernstein form as described in equation (1) to determine the surface points, with $B_i^n(t)$ being the well-known Bernstein polynomial as defined in equation (2) and $\mathbf{P}_{i,j}$ representing the associated control points.

$$S(u,v) = \sum_{i=0}^{m} B_i^m(u) \sum_{j=0}^{n} B_j^n(v)\, \mathbf{P}_{i,j}, \tag{1}$$

$$B_i^n(t) = \frac{n!}{i!(n-i)!} t^i (1-t)^{n-i} \tag{2}$$

In addition, to assure correct reflection mapping, we need to compute the surface normals, which can be derived by the partial derivatives defined in equations (3) and (4).

$$\frac{\partial S(u,v)}{\partial u} = m \sum_{j=0}^{n} B_j^n(v) \sum_{i=0}^{m-1} B_i^{m-1}(u)[\mathbf{P}_{i+1,j} - \mathbf{P}_{i,j}] \tag{3}$$

$$\frac{\partial S(u,v)}{\partial v} = n \sum_{i=0}^{m} B_i^m(u) \sum_{j=0}^{n-1} B_j^{n-1}(v)[\mathbf{P}_{i,j+1} - \mathbf{P}_{i,j}] \tag{4}$$

The surface normal N at (u,v) can then be computed as:

$$N(u,v) = \frac{\partial S(u,v)}{\partial u} \times \frac{\partial S(u,v)}{\partial u} \tag{5}$$

Finally, we compute the reflection vector $R(u,v)$, which we use as the input vector to lookup the cubemap.

$$R(u,v) = S(u,v) - 2(N(u,v) \cdot S(u,v))N(u,v) \tag{6}$$

An alternative approach for deriving $S(u,v)$ and $N(u,v)$ is the recursive deCasteljau algorithm, which we are not describing here. The concept as well as a more general discussion about NURBS and Bézier patches can be found in [23].

3.2. Flexible Camera

As we use a NURBS surface as our *Flexible Film* pendant, we need a more intuitive way to define the cameras parameters than by just specifying the corresponding control points. Therefore, we included an additional abstraction layer and developed a camera model, which is set above the NURBS surface and computes the required control points. This approach is also advantageous for design an API, which can be later integrated into a game engine or other applications and used to control the camera model. This camera model is still work in progress, but we believe the results are promising already. Figure 3 shows a preliminary version of the camera interface.

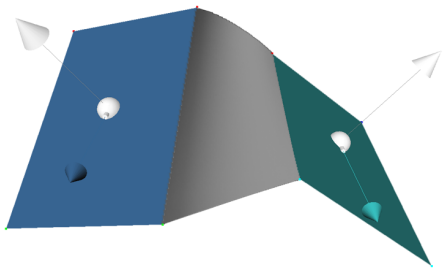

Figure 3: *The camera model.*

At this point of development, our camera model consists

of a set of standard, single perspective cameras that are connected by a flexible mesh. Each camera has its own parameters, like size, orientation and direction, that can be intuitively controlled by our interface, see Figure 3. This allows the control of the NURBS surface by just defining the associated cameras and can easily be integrated into an existing application, like a game engine. One limitation of the current model is the fixed size of the connecting mesh. Ideally, the camera model would support not only a flexible, but also a rubber like surface, so the film can be stretched over the scene, intuitively describing which parts of the film are warped and which are normal. Refereing back to Glassner's drawings, Figure 2, this technique would support a rendering in this manner.

From the just introduced model two problems arise, which we are discussing in the next two sections. The first problem, the *Stitching Problem*, is the correct computation of the patches connecting each camera view, and the second, the *Domain Problem*, is a change in the domain of the NURBS patch describing the Flexible Film.

3.2.1. Stitching Problem

To allow a smoother transition between the single cameras, additional NURBS patches are needed that stitch the existing cameras together. To simulate a "continuous as possible appearance" of the Flexible Film surface, at least C^1 continuity is required. This can be achieved by applying constraints on the appropriate control points. As illustrated in figure 4, only the control points on the border between the two patches ($\mathbf{a_{m,j}} = \mathbf{b_{0,j}}$) must be shared for C^0 continuity. If C^1 continuity is needed, an additional row of control points must be shared by each of the two patches: $\mathbf{a_{m-1,j}}$ and $\mathbf{b_{1,j}}$, which in addition have to be collinear with $\mathbf{b_{0,j}}$.

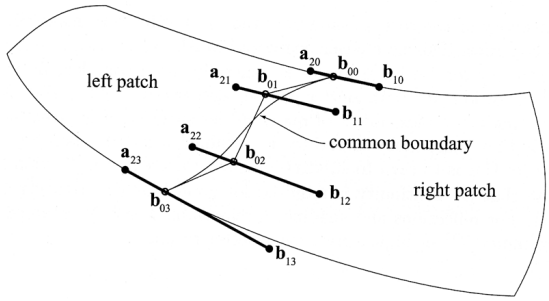

Figure 4: *Stitching together two NURBS patches. (from [2])*

3.2.2. Domain Problem

As lined out in the beginning of the section, a rectangle (GL_Quad) is drawn in full screen size and used to sample the Flexible Film's surface. This procedure is not possible if the Flexible Film is composed of a set of NURBS patches, since each single NURBS patch is defined in the $(0,0)-(1,1)$ domain. Our current solution to this problem is to restrict the "up" vector of the cameras to always point in the *y* direction. This prevents any patch to be non-aligned with the screen axes and we simply need to draw a rectangle for each patch at the appropriate screen positions defined by the cameras. A more elegant solution would be a normalization of the Flexible Film's surface to completely lie within $(0,0)-(1,1)$.

4. Realtime Implementation

In our current implementation we use Coin3D [16] as scenegraph and the Cg language for shader programming. Because the current version of Coin3D did not support any mechanism for GPU shader integration, we decided to implement the necessary OpenInventor nodes ourself and contributed them to Coin. They are expected to be an integral part in the next release. Unfortunately, as only static cubemaps are momentarily supported by Coin, we could not extend our current implementation to integrate dynamic changes of the scene or camera movements. We are currently working on a solution to this problem, but have also written a simple Glut application to verify the feasibility of this approach. We found the dynamic changing of the cubemaps easy to realize and computationally not to expensive.

In the following, we describe different implementations of a Bernstein polynomial based NURBS surface sampling as described in Section 3.1. Although we additionally implemented the alternative deCasteljau algorithm, as used in the Mesa [22] implementation for the automatic generation of NURBS normals, we will not discuss its details here. All shader programs are written in the Cg language.

4.1. Straight-Forward Bernstein

The overall running time of the rendering system is mainly affected by the complexity of the fragment shader used. Therefore, we decided to implement the NURBS with 4×4 control points, as this seems to be a good balance between quality issues and speed. Due to the simplicity of a straightforward implementation, we are not going into details here, and refer to the next section where we discuss several optimizations to increase the rendering speed.

4.2. Optimized Bernstein

As the equation (1) is separable, each of the Bernstein polynomials $B_i^4(u)$ and $B_j^4(v)$ can be computed independently as described in equation (2). The appropriate fragment shader code is shown in listing 1, where *u* and *v* are the incoming texture coordinates from the rectangle.

```
float bernU[4] = {1.0f, 3.0f, 3.0f, 1.0f};
float bernV[4] = {1.0f, 3.0f, 3.0f, 1.0f};
float uu       = 1.0f - u;
float vv       = 1.0f - v;

for (i=0; i<4; i++) bernU[i] *= pow(u, i) *
                                pow(uu, 4-i);
for (j=0; j<4; j++) bernV[j] *= pow(v, j) *
                                pow(vv, 4-j);
```

Listing 1: *Computation of B_i^4 and B_j^4*

According to equation (1) the surface point $S(u,v)$ can then be computed using `bernU[]` and `bernV[]` as shown in listing 2, where `ctrlPoints[]` represents the 16 incoming control points.

```
float3 point = float3(0.0f, 0.0f, 0.0f);
for (j=0; j<4; j++)
  for (i=0; i<4; i++)
    point += (bernU[i] * bernV[j]) *
             ctrlPoints[i*4+j];
```

Listing 2: *Computation of $S(u,v)$*

For the surface normal $N(u,v)$, we need to compute the partial derivatives according to the equations (3) and (4). Since both implementations are conventionally equal, we only describe our implementation of equation (3). The appropriate fragment shader code is shown in listing 3. As before, we can use the separability property of equation (3). Since we can reuse $B_j^4(v)$ we only need to compute $B_i^3(u)$, which is done in the first three lines of the shader.

```
float bernDU[3] = { 1.0f, 2.0f, 1.0f };
for (i=0; i<3; i++)
  bernDU[i] *= pow(u, i) * pow(uu, 3-i);
float3 du = float3(0.0f, 0.0f, 0.0f);
for (j=0; j<4; j++)
  for (i=0; i<3; i++)
    du += (bernV[j] * bernDU[i]) *
          (ctrlPoints[(i+1)*4+j] -
           ctrlPoints[i*4+j]);
```

Listing 3: *Computation of $\frac{\partial S(u,v)}{\partial u}$*

Finally, we can put together all code fragments to obtain a working Cg fragment shader (see listing 4). For the computation of $R(u,v)$, as described in equation (6), we simply exploit the Cg function `reflect()`.

4.3. Cached Bernstein

Although the optimized Bernstein implementation from the previous section already provides a better performance, compared to the straight-forward implementation, there are still some computations which can be saved.

Much time of the algorithm is spend on computing the $B_i^n(t)$ terms, which typically do not change from frame to frame and more importantly, they are independent of the control points. By storing `bernU[]`, `bernV[]`, `bernDU[]`, and `bernDV[]` in one dimensional textures (sampler1D), the computation can be swapped out in a one time preprocessing step on the CPU. The computed data is simply loaded as 1D textures and accessed through the evaluation of the fragment code.

```
float4 main(in Input IN,
            uniform float3 ctrlPoints[16],
            uniform samplerCUBE cubeMap)
           : COLOR
{
  float u  = IN.tex0.x;
  float v  = IN.tex0.y;
  int i, j;

  // computation of bernU/bernV (Listing 1)
  // computation of 'point'     (Listing 2)
  // computation of 'du'        (Listing 3)
  // computation of 'dv'

  float3 norm = normalize(cross(du,dv));
  point = reflect(normalize(point), norm);
  return texCUBE(cubeMap, point);
}
```

Listing 4: *Complete fragment shader*

5. Results

Although, we present work in progress, some preliminary results are available, which we would like to share and discuss in this section. Additional information about this project, as well as more examples can be found on our website[†]. We will update this website as soon as possible and hopefully present the first animations shortly.

As our method is a mixture between image based and direct rendering, it is easily possible to use either artificial, computer generated data sets, or real-world photographs as input. All computations were performed on a Linux computer running Suse 8.1 with the latest Linux diver for nvidia based graphics hardware (driver version 66.29). The computer is equipped with an AMD Athlon 860 MHz with 384 MB of Main Memory and an nvidia QuadroFX 2000.

The first example shows some kind of a super fisheye lens that displays a complete scene overview, Figure 5. Although the room is distorted, one can easily recognize the walls, the windows and the material it is build off. Examples were these type of images are useful are scene overviews in various applications. In order to provide a smooth transition of

[†] `http://isgwww.cs.uni-magdeburg.de/~spindler/wiki/cubism`

Screen Resolution	Normal Computation (Y/N)	Simple NURBS	deCasteljau Implementation	Optimized Bernstein
1024x768	Yes	-	8.775	8.568
1024x768	No	7.939	10.775	20.771
800x600	Yes	-	13.997	13.837
800x600	No	13.014	15.318	30.571
640x480	Yes	-	20.234	21.331
640x480	No	20.218	24.277	51.580

Table 1: *Performance Results*

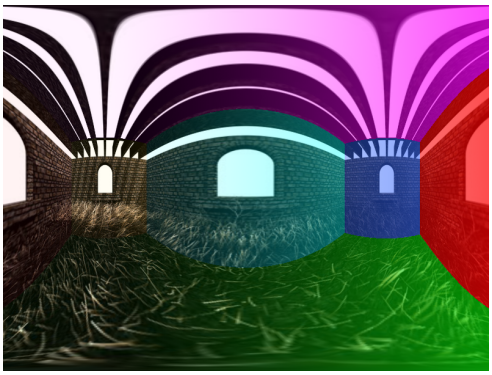

Figure 5: *Scene Overview. (Cubemap from [29])*

the normal screen to this *full screen*, an animation should be used, which changes the NURBS control points and transforms the camera surface into a fisheye lens.

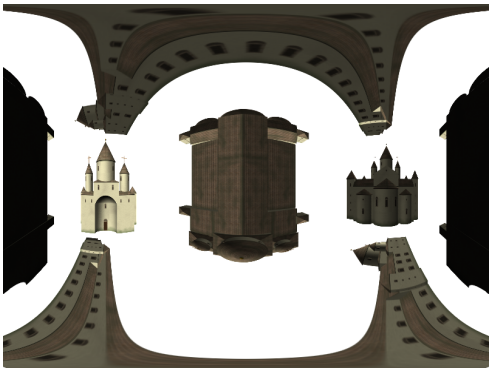

Figure 6: *Object Cubism.*

The next example, Figure 6, is a composition of different views of a virtual reconstruction of an ancient building [10]. The images are rendered with 3D Studio MAX using six cameras that captured the building from all sides. This is an example of object cubism, in which several viewpoints of an object are composed into one single image.

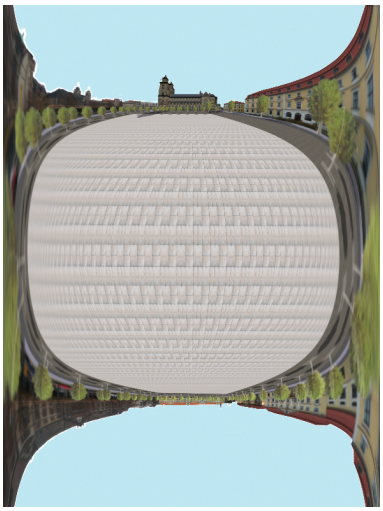

Figure 7: *Fisheye View.*

The last example, which is similar to Figure 1, pictures the main place and the cathedral of our city, Figure 7. This scene was also modelled with 3D Studio MAX and displays a view which was often used in the antiques to draw maps and landscape overviews. The main center is drawn in focus while the surrounding objects vanish.

5.1. Quantitative Results

In this section we want to compare the performance results of the implemented techniques. We have in total four slightly different algorithms: the simple NURBS implementation, the deCasteljau, an optimized Bernstein and a cached Bernstein implementation. Unfortunately, we are not finished yet

with the fully testing of the cached Bernstein implementation, but the preliminary results we have so far show tremendous speed improvements. Although these speed gains are only valid as long as the control points are unchanged, but in most cases only one or few control points are changed simultaneously.

Table 1 shows the performance results for the straightforward NURBS implementation, as well as for the deCasteljau algorithm and the optimized Bernstein implementation. As can be seen from these results, that the implemented optimizations really pay off and that the achieved rendering speed is sufficient for an integration into a game engine. Some speed drawbacks will occur in future version, if dynamic cubemaps are used and additional computations have to spend on the cubemap rendering.

6. Summary and Future Development

We have presented a new technique for the realtime rendering of cubist-style images. The rendering performs in realtime on advanced graphics hardware, with an average frame rate of about 20 to 30 frames per second. We experimented with varying implementations of NURBS surfaces and provided some optimization techniques to gain additional speed improvements. We showed that the proposed technique is useful for real world photographs, as well as for computer generated images. The presented method can be applied to either object or scene multi-perspective rendering.

The possibilities for future improvements of the presented technique are manifold. Concerning the realtime rendering, the next steps are the extension of the algorithm to use dynamic cubemaps and therefore an animated camera model. This requires an extension of the existing camera and adaptation of the user interface to allow dynamic scene settings. In addition, as the cubemaps are sampled throughout the rendering process, aliasing or pixel artifacts might occur. The quality of the current implementation is good enough for realtime rendering, but to achieve a higher image quality, cubic interpolation can be implemented and used instead of the hardware supported linear filtering.

Furthermore, the presented techniques have to be evaluated within real applications. Here, we will focus on two applications: the integration into a computer game engine to explore the possibilities for story telling and game play and to develop a brief story for a short film, which is based on the main concept of our multiple perspective camera model. In both applications, the flexible camera model will only be employed in cases where the utilization is beneficial. As our camera model allows the simulation of a conventional perspective camera system, these scene transition should be easy to implement.

References

[1] Maneesh Agrawala, Dennis Zorin, and Tamara Munzer. Artictic Multiprojektion Rendering. In *Eurographics Rendering Workshop*, pages 125–136, 2000.

[2] Thomas Akenine-Möller and Eric Haines. *Real-Time Rendering, 2nd Edition*. Springer, 2002.

[3] Joachim Böttger. Darstellungen von Betrachtungsvorgängen. Master's thesis, Otto-von-Guericke University Magdeburg, Department of Simulation and Graphics, 2003.

[4] Veronique Bourgoin, Nathalie Farenc, and Marc Roelens. Creating special effects by ray-tracing with non classical perspectives. Technical Report ENSM.SE 1995-13, Laboratoire d'Image de Synthese de Saint-Etienne, 1995.

[5] M. S. T. Carpendale, D. J. Cowperthwaite, and F. D. Fracchia. Extending Distortion Viewing Techniques from 2D to 3D Data. *IEEE Computer Graphics and Applications, Special Issue on Information Visualization*, 17(4):42–51, 1997.

[6] S. H. Chu and C. L. Tai. Animating Chinese Landscape Paintings and Panorama using Multi-Perspective Modelling. In *Computer Graphics International*, Hong Kong, 2001. IEEE Press.

[7] Patrick Coleman and Karan Singh. RYAN: Rendering Your Animation Nonlinearly projected. In *3rd International Symposium on Non-photorealistic Animation and Renderinr*, 2004.

[8] J. P. Collomosse and P. M. Hall. Cubist Style Rendering from Photographs. *IEEE Transactions on Visualization and Computer Graphics*, 2002.

[9] Wikipedia Community. Wikipedia Online Encyclopedia. `http://www.wikipedia.org/`, 2005.

[10] Bert Freudenberg, Maic Masuch, Niklas Röber, and Thomas Strothotte. The Computer-Visualistik-Raum: Veritable and Inexpensive Presentation of a Virtual Reconstruction. In Stephen N. Spencer, editor, *VAST 2001: Virtual Reality, Archaelogy, and Cultural Heritage*, pages 97–102; 365–366. ACM, 2001.

[11] Bert Freudenberg, Maic Masuch, and Thomas Strothotte. Walk-Through Illustrations: Frame-Coherent Pen-and-Ink-Style in a Game Engine. In *Computer Graphics Forum: Proceedings Eurographics*, 2001.

[12] Andrew Glassner. Andrew Glassner's Notebook: Digital Cubism, Part1. *IEEE Computer Graphics and Applications*, 24(3):82–90, May/June 2004.

[13] Andrew Glassner. Andrew Glassner's Notebook: Digital Cubism, Part2. *IEEE Computer Graphics and Applications*, 24(4):84–95, July/August 2004.

[14] Andrew S. Glassner. Cubism and Cameras: Free-form Optics for Computer Graphics. Technical Report MSR-TR-2000-05, Microsoft Research, 2000.

[15] Youichi Horry, Ken-ichi Anjyo, and Kiyoshi Arai. Tour Into the Picture: Using a Spidery Mesh Interface to Make Animation from a Single Image. In *International Conference on Computer Graphics and Interactive Techniques*, pages 225–232, 1997.

[16] Systems in Motion. Coin3D Website. http://www.coin3d.org, 2005.

[17] Tobias Isenberg. Expressive Distorsion of Strokes and 3D Meshes. unpublished, 2003.

[18] Allison W. Klein, Peter-Pike J. Sloan, R. Alex Colburn, Adam Finkelstein, and Michael F. Cohen. Video Cubism. Technical Report MSR-TR-2001-45, Microsoft Research, 2001.

[19] Helwig Löffelmann and Eduard Gröller. Ray Tracing with Extended Cameras. Technical Report TR-186-2-95-06, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 1995.

[20] D. Martin, S. Garcia, and J. C. Torres. Observer Dependent Deformations in Illustration. In *1st International Symposium on Non-photorealistic Animation and Rendering*, 2000.

[21] Maic Masuch and Niklas Röber. Game Graphics Beyond Realism: Then, Now and Tomorrow. In Marinka Copier and Josst Reassens, editors, *Level UP: Digital Games Research Conference*. DIGRA, Faculty of Arts, University of Utrecht, 2004.

[22] Brian Paul. Mesa3D Website. http://www.mesa3d.org, 2005.

[23] Les Piegl and Wayne Tiller. *The NURBS Book*. Springer, 1996.

[24] Bernhard Preim. *Interaktive Illustration und Animation zur Erklärung komplexer räumlicher Zusammenhänge*. PhD thesis, Otto-von-Guericke University Magdeburg, Department of Simulation and Graphics, 1998.

[25] Andreas Raab. *Techniken zur Exploration und Visualisierung geometrischer Modelle*. PhD thesis, Otto-von-Guericke University Magdeburg, Department of Simulation and Graphics, 1998.

[26] Paul Rademacher. View-Dependent Geometry. In *ACM SIGGRAPH 99*, pages 439–446, 1999.

[27] Paul Rademacher and Gary Bishop. Multiple-Center-of-Projection Images. In *ACM SIGGRAPH 98*, 1998.

[28] Martin Reinhart. tx-transform Online resources. http://www.tx-transform.com/, 1998.

[29] Sascha Stojanov. Codemonsters, 2005. http://www.codemonsters.de.

[30] Ubisoft. XIII, 2003. PC.

[31] Scott Vallance and Paul Calder. Multi-Perspective Images for Visualization. In *Pan-Sidney Area Workshop on Visual Information Processing*, 2002.

[32] Daniel N. Wood, Adam Finkelstein, John F. Hughes, Craig E. Thayer, and David H. Salesin. Multiperspective Panoramas for Cel Animation. In *ACM SIGGRAPH 97*, 1997.

[33] Geoff Wyvill and Craig McNaughton. Optical Models. In *CGI*, 1990.

[34] J. Yu and L. McMillan. A Framework for Multiperspective Rendering. *15th Eurographics Symposium on Rendering*, 2004.