# Hardware Accelerated Motion Blur Generation

Clement Shimizu, Amit Shesh, Baoquan Chen

{clement | ashesh | baoquan }@cs.umn.edu
University of Minnesota at Twin Cities

**Abstract**
*Motion blur occurs in photography by the motion of objects during the finite exposure time that the camera shutter remains open for to record the image on film. The traditional method of rendering a motion blur with a computer is to render the scene at many discrete time instances in every frame. In this paper, we present an efficient motion blur generation method that leverages modern commodity graphics hardware. Our method avoids rendering the entire complex scene many times per frame. It first renders the scene into a texture, next renders the optic flow, created based on object transformation, to a vector field texture. The scene texture is finally efficiently blurred according to the vector field using texture-mapping hardware to do a piecewise iterative line integral convolution. Though our method uses vertex velocities to calculate image pixel velocities, the line integral convolution is performed on an image, making our method largely independent of scene complexity.*

**Keywords:**
*Motion Blur, Line Integral Convolution, Image Warping, Offset Texture, Vertex Shader, Cg, Optic Flow*

## 1. Introduction

In the virtual world of the computer, at discrete time instances, objects are at discrete locations. When rendering the virtual world to a display device the world is captured at one singular instant in time. As a result the blur a camera or a human eye sees when viewing a rapidly moving object is absent. The blurring of the image taken by a camera due to motion can be attributed to the exposure of the camera film to light for an amount of time in which an object moves a certain distance.

The traditional method of rendering a motion blur with a computer is, for a single frame, to render the scene at many discrete time instances, average these renderings and output to the display device. To save space, the rendering and averaging can be done in hardware using the accumulation buffer [5]. However the quality of the final output of each frame depends on the number of renderings that are combined to make it. Since the whole scene must be rendered multiple times, this method does not scale well. If the number of polygons in the scene is large, rendering the scene multiple times will lower the frame rate. If the number of times a scene is rendered per frame is too low for the amount of motion happening in the scene, then instead of a smooth motion blur, the rendering produces ghosting or double vision as seen in figure 1. This effect is called *temporal aliasing*. A virtual world with fast motion may need to be rendered up to 20 times or more per frame to eliminate ghosting. The 3dFx Voodoo5 6000 had 4 gpus and frame buffer memory segments that could do a 4-step motion blur in parallel. Although this can be done in parallel, there are practical and intelligent approaches of performing the same more efficiently and in less demanding ways.

**Figure 1:** *Ghosting or temporal aliasing effect when the motion blur of a fast moving object is generated using accumulation buffer but large time interval between multiple renderings.*

In this paper, we present an efficient motion blur generation method which leverages modern commodity graphics hardware. Our method avoids rendering the entire complex scene many times per frame. It first renders the complex scene to a texture, and then renders the optic flow created based on object transformation to a vector field texture. The scene texture is finally efficiently

University of Minnesota Computer Science Department Technical Report 2003-01.

blurred according to the vector field using texture mapping hardware by performing piecewise iterative line integral convolution [2]. After obtaining the image pixel velocities from the vertex velocities, all operations including the line integral convolution are done on an image, making our method largely independent of scene complexity.

The organization of this paper is as follows. Section 2 introduces previous work. Section 3 gives an overview of our algorithm; and an efficient implementation of the same is explained in Section 4. Finally we present results in Section 5 and offer discussions and future work in Section 6.

## 2. Previous Work

The motion blur effect has often been addressed in combination with solving the spatial aliasing problem. The mathematical expression for the intensity of a pixel I(x,y) on the screen is generalized by the equation:

$$I(x,y) = \iint_{\Omega T} R(\varpi,t)L(\varpi,t)\,dtdw \qquad (1)$$

where L(ω,t) is the incoming luminance function and R(ω,t) is the reconstruction filter [13]. In this equation smooth motion is reconstructed from a discrete set of renderings (images) over time.

### 2.1. Solving Temporal Visibility and Shading:

Sung *et al.* [13] approach this problem in a very general framework by solving the spatial-temporal aliasing problem. Their approach, though generic in nature, demands significant hardware, both as memory and processing power. The authors solve the two aliasing problems separately by decomposing equation (1) into time and spatial domains, and solving the spatial temporal visibility and spatial temporal shading problem separately, by observing pixel coverage per object per time sample and integrating over time and space.

### 2.2. Synchronization of Frame and Texture Frequency:

There are some approaches, which solve the problem by preventing its cause. Norton *et al.*'s method [9] limits the frequency of texture functions to the pixel-sampling rate [13]. This essentially matches the rendering frequency with the motion frequency and removes jerkiness. However, this method requires prior knowledge of motion.

### 2.3. Post-Processing Algorithms:

Post processing algorithms work in object space or image space, keeping the interference of the motion blur algorithm in the object-rendering algorithm to a minimum.

#### 2.3.1. Motion Blur by Time Convolution:

Potmesil and Chakravarty [11] promulgate an approach of producing motion blur by time convolution of the normal image with the motion function. This approach works in image space, and is also capable of producing images with moving and non-moving objects, by post-processing the image of moving objects and blending it with the image of non-moving objects. A Fourier transform approach for time convolution is taken, which is difficult to implement in current commodity graphics hardware, although iterative time convolution in real time is feasible.

#### 2.3.2. Line Integral Convolution (LIC):

Convolution along a vector in the direction of motion produces directional blur. This is the approach chosen by Cabral and Leedom [2]. A line integral is used instead of a piecewise linear approximation to express the direction of motion. The vector field is created with this line integral, and the normal image is convoluted per pixel in the direction of the corresponding vector in the vector field. This approach, however, produces good effects only if a dense, per-pixel vector map is available, which is assumed in [2] as a preprocessing task.

#### 2.3.3. Blurring by Image Interpolation:

Chen and Williams [1] present an image synthesis method through view interpolation – a process of interpolating on a per pixel basis to produce novel images. They discuss motion blur as one of the many effects which can be produced by their approach. Their method too, requires a dense per-pixel vector field.

The approach adopted in this paper is inspired by the last two methods mentioned above. Thus, our approach works in image space, and is a post-processing step. Independently, Green [4] described a similar approach that uses image warping by offsetting a texture to produce motion blur and uses fragment shaders.

## 3. Algorithm Overview

Our method is based on iteratively warping the image of the object to be blurred and consecutively blending it with the image produced by the earlier iteration. As will be obvious in the implementation section, the algorithm operates in image space. The algorithm is divided into the following three steps and is summarized in figure 2:

**Step 1: Creating Offsets:**
The scene is assumed to be expressed as a collection of polygons. As the scene is rendered, a per-vertex displacement vector field is computed. This field stores the vertex offset in the form of a three-dimensional vector, which is used to warp the object. This vector is determined by two factors. Firstly, it is determined by the relative directions of the vertex normal and the direction of motion, because if a vertex is displaced along its normal, then it has to be maximally warped. This is so because motion in object space along the normal translates to more movement of its projection in image space. Secondly, the offset is determined by the speed of motion, which in this case, is indicated by the magnitude of the velocity vector itself. Therefore, at every vertex, the vector offset for warping is given by

$$W = (n \cdot v)V \tag{2}$$

where $n$ and $v$ are the normal and velocity unit vectors and $V$ is the total velocity vector.

**Step 2: Performing Line Integral Convolution:**
The object is next warped on a per-vertex basis using the offset vector field that was obtained in step 1. We start with the maximum warp possible, and then move towards the actual final position of the object. The maximum warp is determined by the relation between the speed of the mouse which is used to move the object in our implementation, and the actual speed it is simulating. This factor is a user-defined parameter. The warped object is then alpha-blended with the original object, and this blended object is used as the original object in the next iteration. Blending is done with warps in directions along and opposite to the vector field, but the latter is done with a lesser scaling factor. The effect is that the trailing edges of the objects are blurred more than the leading, producing the typical blur trail. The alpha value for blending increases as the amount of warp decreases. This is intuitive because as the shape of the warped object gets closer to the original shape of the object, pixels that are a part of the warped object have a greater chance of being a part of the object in its final position. This iterative blending performs line integral convolution, along the path of motion in a heuristic manner. The extent and quality of blur depend upon the number of iterations over which LIC is carried out, and also on the difference between consecutive warps.

**Step 3: Rendering the Final Blurred Image to Screen:**
The final blurred object is now rendered to the screen to show the motion-blurred object.

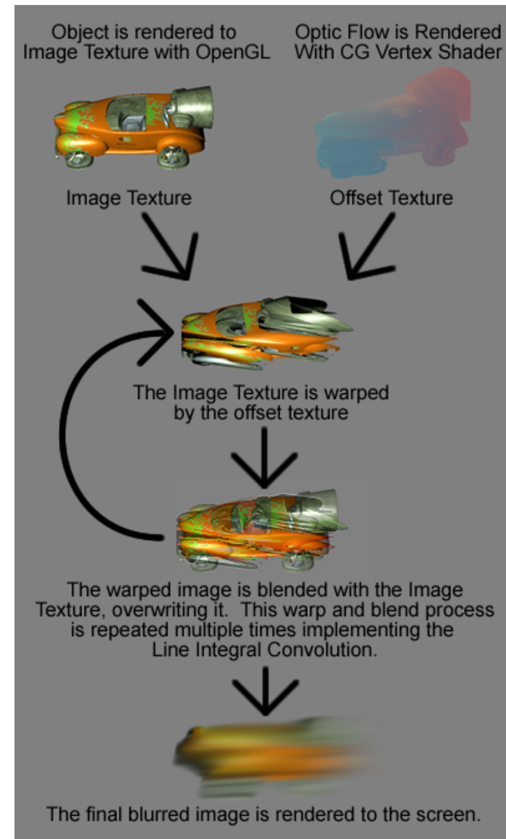



**Figure 2.** *An overview of the motion blur algorithm, with screenshots taken at every step of the algorithm.*

## 4. Implementation:

The main motivation of our work is to make the implementation as efficient as possible, using current commodity graphics hardware, and deferring as many operations as possible to the GPU during the conventional image rendering. We have used the *NVIDIA GeForce Quadro* 4 graphics card for our implementation; the hardware features such as vertex and pixel shading capabilities that we use are common to other graphics cards. For demonstration purposes, we embedded our implementation into code obtained from [10]. This was done solely to have a good object offering flexible manipulations as a starting point. Our candidate model, the NVIDIA Rocket Car, has a polygon count of about 29000. The static background texture has been taken from [6].

The current implementation blurs an object against a background image, which is available as a texture. This method can be generalized by making separate sets of moving and non-moving objects in a preprocessing

stage, and then merging them together in the blending stage, as suggested in [11].

The vertex displacements are determined by a vertex shader and are rendered to the framebuffer as pixel displacements. Since the shader cannot render negative offsets, we shift the range from (-1,1) to (0,1). Now, the algorithm warps the object based on relative directions of the velocity vector and the vertex normal. This may violate visibility constraints if the object has non-manifold parts. This is because, when the dot product $(n \bullet v) < 0$, no offset is generated. In non-manifold objects where two normals face each other, an inner face may relatively pop out, giving an incorrect vector field. To cure this anomaly, the vector field is rendered twice, in warped and non-warped versions, and the union is taken. These shifted vertex displacements are then read into an array of RGB values, and are shifted to (-0.5,0.5). The R and G values signify the *x* and *y* offsets respectively, while the B component is neglected. To do this, we rearrange the RGB values to R-G values. They are then copied to a texture of DSDT_NV format. This texture format stores texture offsets (ds,dt) at texel location (s,t).

**Figure 3**. *A visualization of the offset vector field, showing red color for displacements along X direction, and green color for displacements along Y direction*.

The original object is rendered to a texture, using a *pixel buffer*. Thus, all operations henceforth are performed on textures in image space. Warping is produced by passing these two textures (original RGB and offset DSDT) to a texture shader. In order to produce a warping effect, the texture shader performs a scaled offset lookup based on the DSDT texture and the warp factor (the scale), in the RGB texture. The RGB texture is then alpha blended with the warped texture to form the new RGB texture for the next warp-and-blend iteration. The alpha function can be set to gaussians or ramps to obtain varying kinds of blur. This blending is done with a pixel buffer as the rendering context, due to which, the blended image automatically becomes available as a texture. Successive lesser warps are produced by manipulating the warp factors in the texture shader. When all iterations of LIC are done, the rendering context is switched back to the framebuffer, and this final texture is simply rendered on it to show the blurred object.

The following table summarizes the various operations and where they are implemented (CPU or GPU):

| **Operation** | **Performed by/on** |
|---|---|
| Generating the vector offsets | Vertex shader (GPU) |
| Adjusting offsets and creating DSDT texture | CPU |
| Rendering object as an image | Pixel Buffer (GPU) |
| Warping | Texture shader(GPU) |
| Blending | Pixel Buffer (GPU) |

For demonstration purposes, some additional computations are performed to render an optic-flow vector field to show the direction of motion as seen by the vertices (see figure 4).

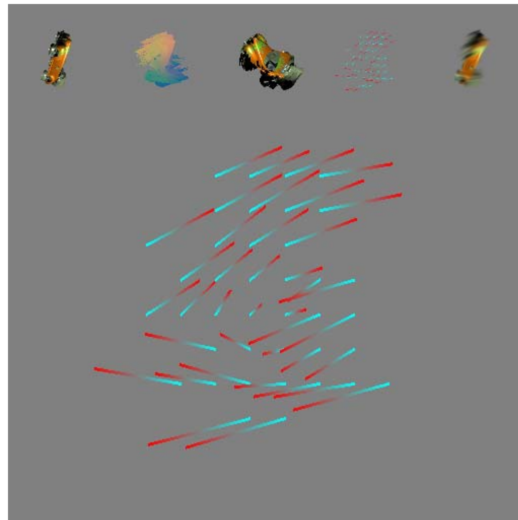


**Figure 4.** *A visualization of the optic flow vector field. The blue start point of the flow shows the original texture coordinate, and the red end point of the flow shows the texture coordinate looked up after applying the offset found in the DSDT texture.*

## 5. Results:

For demonstration purposes, our implementation produces a main picture and five thumbnails, as shown in figures 5 and 6.

From left to right, the thumbnails show: the original object, the offset-vector field (with red color for motion in X direction and green color for motion in the Y direction for visualization purposes), the object warped by the offset field, the optic flow vector field and the final blurred object.

**Figure 5.** *Screenshot showing a blurred car moving in the horizontal direction (pure translatory motion)*

Figure 5 shows a screenshot of the motion blur produced by our implementation for pure translatory motion. The "red" offset vector field in thumbnail 2 shows that this is a motion only along X axis, as is further evidenced by the optic flow vector field in thumbnail 4. The user defined parameter for actual speed simulation (refer to Section 3.2) is set to simulate high speed, justifying the significant blur.

Figure 6 shows a screenshot of the motion blur produced for rotational motion. The varying colors in the vector field show the various directions in which each vertex moves. From the "trail" of the blur, it is immediately apparent that the rotation is clockwise.

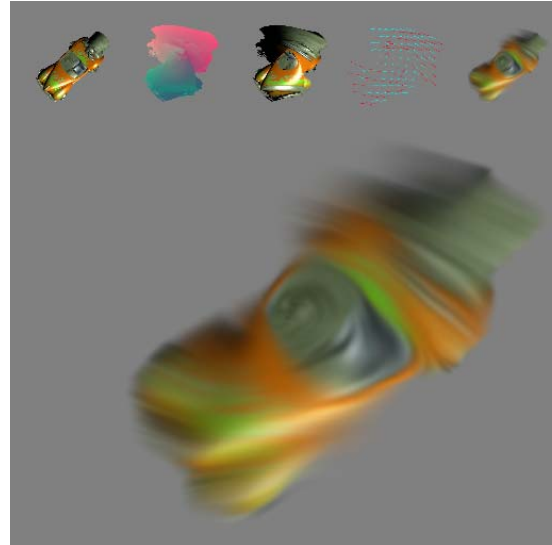


**Figure 6.** *Screenshot showing a blurred car rotating about its central axis (rotational motion).*

**Figure 7.** *Screenshot showing the blurred car against a static background.*

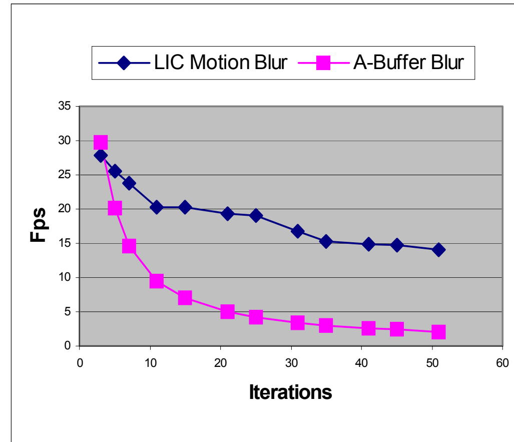
**Quantitative comparison:**

We tested the implementation of our algorithm with a few practical methods of producing motion blur effect. The test model, again, is the "NVIDIA Rocket Car" and the implementation, without any effects, runs at a frame rate of 60fps. The resolution was maintained at 512 x 512 pixels.

The practical method, as mentioned in Section 1 is to break a time interval into several sub-intervals and accumulating them using an accumulation buffer. The speed of this method can be competitive only if the accumulation buffer is implemented in hardware. A routine implementation using the accumulation buffer quickly produces very poor frame rates (less than 1) as more and more renderings per frame are done. An alternative method is to render only one image per frame, and accumulate it with the current contents of the accumulation buffer, thus effectively maintaining a "running sum" of all images drawn. This method produces 49fps for the candidate model, and scales very well. However, there is noticeable temporal aliasing and the ghosting images are very distracting.

There are very few implementations of commodity graphics hardware that support an accumulation buffer, an example of which is the 3dfx Voodoo T-buffer. Examples of motion blur produced by the T-buffer can be seen at [12], which also show the temporal aliasing effects illustrated in figure 1. In general, this method of rendering an object repeatedly does not scale well with the number of iterations of accumulation, nor with the polygon count of the scene because the object has to rendered more and more number of times, which lowers the frame rate to non-interactive values. However, we implemented a variant of this method on the GeForce Quadro 4 graphics card to serve as a benchmark. In this implementation, we rendered the car to texture several times in a frame and used alpha-blending to merge them. This essentially retains the flavour of the conventional accumulation buffer motion blur method. The comparison of this implementation with our warping-and-blending approach produced the following results:

| Iterations per frame | 03 | 05 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| A-buffer frame rate | 29.7 | 20.1 | 9.4 | 7.03 | 5.04 | 4.25 | 3.43 | 3.04 | 2.61 | 2.37 | 2.01 |
| Our frame rate | 27.8 | 25.6 | 20.3 | 20.3 | 19.3 | 19.1 | 16.8 | 15.24 | 14.8 | 14.7 | 14.0 |




The table shows the frame rates and quality of the images produced by the variant of the accumulation buffer implementation and our method in comparison. We have chosen the visible distance between ghosts as the metric for quality of blur. An image where images at discrete sub-intervals are visually inseparable shows a good blur.

From the table and the adjoining graph, it can be clearly seen that our method scales very well as the number of iterations (deciding the quality of blur) increase. This is attributed to two factors; the quasi-image-space nature of our method and the hardware acceleration.

**Limitations:**

There are some limitations to our current implementation.

Firstly, the static background texture is a part of the normal image over which the line integral convolution is done. Though this effect is not very noticeable, this blurs the background image in the vicinity of the moving object.

Secondly, we follow the method of separating all moving and non-moving objects in a scene and merge them in the last step of our algorithm. This potentially leads to either a merger of vector fields, which could produce incorrect motion blurs of two objects overlapping each other, or a separate pass for each moving object.

As our method is image-based, the frame rate will lower if resolution of the scene is increased.

## 6. Future Scope

We feel our algorithm and implementation can be further enhanced to produce more realistic and efficient motion blur.

The current implementation starts from the maximum warp and progresses towards the true final position of each pixel. A forward-warp approach could be implemented, which reverses this direction. This approach could minimize the problem with non-manifold objects discussed in Section 4. Specifically, as the iterations proceed, we would move more towards the more blurred part, with decreasing values of alpha. As such, the warping, which was causing some inner surfaces of non-manifold object parts to pop out, would contribute lesser and lesser to the total blurred image.

There are two extensions which could make our implementation even more efficient. Firstly, three texture offset lookups could be done in the texture shader, using three consecutive scaling factors on the same pair of textures, offset and RGB. Their blending can be done in a register combiners' stage. This essentially decreases the number of passes of LIC to get the desired extent of blur. Secondly, we noticed that the step of forming the DSDT texture by manually downloading it from GPU memory, then shifting the offsets and uploading it back to texture memory forms the bottleneck of our implementation. Though innovative texture formats like DSDT_NV and its variations are provided, general register instructions like *expand* cannot be used with them. We look forward to future, more flexible implementations of these formats. We also look forward to any future, more efficient methods of generating a DSDT texture using RGB textures, which can currently be efficiently generated using vertex shaders.

Generating motion blur by an image warp-and-blend approach forms the heart of our idea and our algorithm. This can be achieved in several ways and different implementations. An alternative implementation to ours is to use a pixel shader to perform the iterative line integral convolution by texture offsets. The current version of OpenGL does not support pixel shaders per se. Future implementations of the CG compiler for the NVIDIA GeForce FX do promise to support more advanced pixel due to which the entire LIC step could be pushed to the pixel shader stage.

## 7. Conclusion

As the power of consumer video cards increases, so does the complexity of virtual worlds. It is important to consider how special effects will scale when applied to scene complexities approaching millions of triangles. Image-based techniques that render the geometry to textures and apply the effects to the textures will become of higher utility, since it may be faster than rendering the entire scene over again. As texture shaders become more flexible and easier to program, we think we will see more advanced techniques where objects are rendered to textures and processed before rendered to the screen.

## 8. Acknowledgements

This work was supported in part by the Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement number DAAD19-01-2-0014. Its content does not necessarily reflect the position or the policy of this agency, and no official endorsement should be inferred. Other support has included a Computer Science Department Start-Up Grant and a Grant-in-Aid of Research, Artistry, and Scholarship from the Office of the Vice President for Research and the Dean of the Graduate School, all from the University of Minnesota.

## 9. References

1. CHEN S. E., WILLIAMS L., "View Interpolation for Image Synthesis", *Apple Computer Inc*.
2. CABRAL B., LEEDOM L. C., "Imaging Vector Fields Using LINE Integral Convolution", *Proceedings of SIGGRAPH 1993,* pp. 264.
3. DIGIBEN., Motion Blur, In *OpenGL Game Tutorials* http://www.gametutorials.com/Tutorials/opengl/OpenGL_Pg2.htm
4. GREEN SIMON, "Stupid OpenGL Shader Tricks", *Game Development Conference 2003*.
5. HAEBERLI P., AND AKELEY K., "The Accumulation Buffer: Hardware Support for High-Quality Rendering", *SIGGRAPH 1990*, pp. 309-318.
6. LIGHT PROBE IMAGE GALLERY http://www.debevec.org/Probes/
7. MCREYNOLDS T., 1998, Advanced Graphics Programming Techniques Using OpenGL, In *Siggraph Courses 1998,* Using OpenGL to perform Line Integral Convolution(LIC) images. HTTP://WWW.SGI.COM/SOFTWARE/OPENGL/ADVANCED98/NOTES/NODE59.HTML
8. MITCHELL, D.P., "Generating Antialiased Images at Low Sampling Densities", *Computer Graphics*, vol. 21, no. 4, pp. 65-72, July 1987.
9. NORTON A., ROCKWOOD A.P., AND SKOLMOSKI P.T, "Clamping: A Method of Antialiasing Textured Surfaces by Bandwidth Limiting in Object Space", *Computer Graphics*, vol. 16, no. 3, pp. 1-8, July 1982.
10. NVIDIA Developers' Website http://developer.nvidia.com
11. POTMESIL M. AND CHAKRAVARTY I., "Modeling Motion Blur in Computer-Generated Images", *Computer Graphics,* Vol. 17**,** no. 3, pp**.** 389-399, July 1983.
12. PREVIEWS OF THE VOODOO 3DFX T-BUFFER http://www.hardwarecentral.com/hardwarecentral/previews/1646/3/.
13. SUNG K., PEARCE A. AND WANG C., "Spatial-Temporal Antialiasing", *IEEE Transactions on Visualization and Computer Graphics*, Vol. 8, April-June 2002

**Figure 8.** *Comparison of the traditional accumulation buffer method and our motion blur method. The number of iterations from left to right is 2 ,4 ,8 and 15. The first row shows the output of the accumulation buffer method and the lower row shows output of our method.*

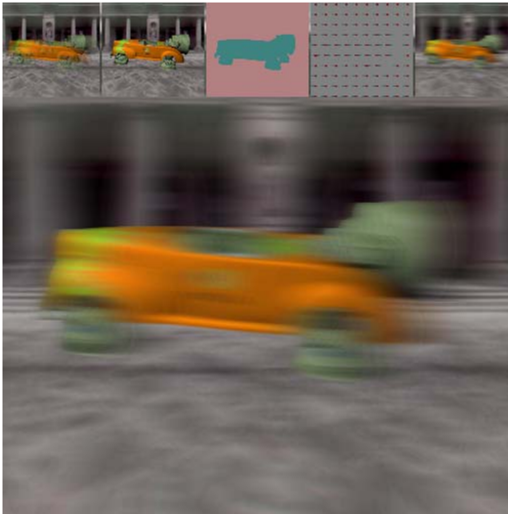


**Figure 9.** *Multiple moving objects in a single Scene. The car moves from right to left while The whole scene moves from left to right.*

**Figure 10.** *Zoom-on the blurred car shows the background in the vicinity of the car blurred with the car, as discussed in Section 5*