Neon Racer: Augmented Gaming

Wolfgang Litzlbauer[*]
Ines Stuppacher[†]
Manuela Waldner[‡]
Markus Weilguny[§]

Digital Media
Upper Austria University of Applied Sciences
Hagenberg / Austria

Abstract

Neon Racer is a multi-user Augmented Reality racing game adapting the simple and powerful gameplay of racing games to an Augmented Reality tabletop setting. The game combines an intuitive and tangible interface with quality content. The active setting for the game is provided by the real world. Physical objects act as collision obstacles and influence the course of the race itself. In this paper we describe the architecture of the game and its compact hardware setup. The collision detection with real objects is explained. Furthermore a novel particle system used in Neon Racer is introduced, which is executed fully on the graphics card.

Keywords: table-top, AR racing game, tangible user interface, face-to-face, GPU particle system,

1 Introduction

In conventional computer games, users are focused on the screen rather than the real world. This reduces the range of actions and social collaboration. Alternate interfaces, such as cameras and microphones, can change the way we look at classical computer games. These novel approaches can help users get more involved in a game quickly. Using intuitive interfaces is also crucial in bringing users together. For years Augmented Reality (AR) research has redefined the possibilities of creating applications for entertainment [1], [6]. AR applications make it possible to enrich virtual games with the social aspects of traditional board games. Players can communicate, exchange objects and abide to a common set of rules [5]. In AR users can share a gaming environment and explore the possibilities of the augmented world together. Games with simple rules, like racing games, are particularly easy to learn and motivating, as long as the rules allow each player to develop his individual style. Players can challenge their own limits while exploring the game world. Players can share both a physical and virtual environment in Augmented Reality enhancing the gaming experience. Neon Racer shows a possibility to boost social interaction in an open environment such as conferences, festivals, shopping malls or museums.

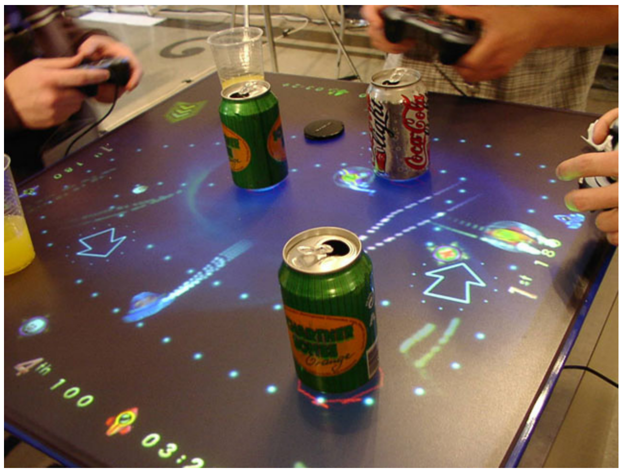

Figure 1: Neon Racer is a multi-user racing game which includes the physical reality as part of the game.

2 The Neon Racer Game

Neon Racer combines a virtual racing game with physical interaction (see figure 1). A benefit of transporting racing games to AR is the ability to include physical objects as part of the game. Neon Racer creates a rich gaming experience by using everyday objects as the setting of a racing game for up to four players. The virtual world contains the players' vehicles, which are controlled with gamepads. Players can use both gamepads and real objects to influence the game. Virtual vehicles collide with real objects, allowing players to move in both the physical and virtual world. The use of two intuitive user interfaces – real objects and gamepads – bridges the gap between virtual and physical interfaces.

The virtual vehicles (see figure 2) are steered by up to

[*]wolfgang.litzlbauer@fh-hagenberg.at
[†]ines.stuppacher@fh-hagenberg.at
[‡]manuela.waldner@fh-hagenberg.at
[§]markus.weilguny@fh-hagenberg.at

Figure 2: The four different vehicles.

four players, who have to maneuver their vehicles through virtual checkpoints. The aim of the game is to score as many points as possible by crossing these checkpoints. A race ends when a time limit has been reached, usually three to five minutes.

The rules of the game are simple and intuitive. Neon Racer does not require more than basic knowledge of computer games. Real objects placed on the course act as obstacles in the game. Players have to maneuver their vehicles past these objects and through the checkpoints. Both users and spectators can move objects around the course, allowing them to contribute to the game itself. Thus usually passive bystanders can actively change the outcome of the race and even take sides. In an exhibition, users used coffee mugs and mobile phones to play, even left packets of handkerchiefs and chocolate bars. Neon Racer is suitable for up to four players but can also be played alone with spectators who manipulate the race course. The simple and cheery nature of the game appeals to a wide range of cultures and ages.

Neon Racer offers several action-packed features such as photon torpedoes, exploding sheep, and turbo-fields. Besides their entertainment value, these extras allow advanced players to engage opponents on a more tactical level. Health-packs make sure that there is always a surprise up someone else's sleeve when things get close.

2.1 Game Development

Neon Racer is based on a simple but powerful racing game engine which can be played on any modern computer. The engine has a modular interface to communicate with a separate obstacle detection system. This system detects the physical objects on top of the table and informs the game if a vehicle is colliding. The obstacle detection is described in detail in section 3.2.

2.1.1 2D-Sprite engine

The core of the game is a multi-purpose 2D-sprite engine. It has been created in OpenGL and supports partial transparency, sprite- and frame-by-frame animation. Each sprite has a position, size and an alpha-value. These values can be smoothly changed over time. The frames of an animation must be stored side by side in one texture. The engine automatically switches between the frames. This enables stunning animations like the explosions in the game. OpenGL shifts the efforts of calculating drawing from the CPU to the graphics card. This is important for proper execution of the extensive obstacle detection system. A novel particle system displays a trail behind the vehicles, caused by their power unit. The particle system is also fully executed on the graphics card and is described in section 4.

2.1.2 Physics engine

An important aspect of the game is the physically correct behaviour of the vehicles. A self-contained physics engine handles the movement and bounciness of the vehicles. Each virtual object in the game, for example a vehicle or a rocket, is represented as a circle. Most entities in the game are round, for example the vehicles have a shield protecting them. This reduces the efforts of detecting a collision. Each object has an individual velocity stored as a direction vector (see figure 3 (1) and (2)). Each frame their positions are recalculated according to their velocity. The objects are moved by applying forces. A force pushes the object in a given direction and increases the velocity.

If two virtual objects collide, a realistic bouncing behaviour is simulated (see figure 3 (3)). A new velocity for the objects is calculated according to the impact angle and the collision strength. The physics engine also considers the real obstacles on the top of the table. Therefore the separate obstacle detection system is asked if a vehicle is colliding with a real obstacle. The system returns a normal of the obstacle's border line if a collision occurs (see section 3.2). This normal is necessary for the calculation of the rebound.

Because Neon Racer takes place in space, the vehicles are weightless. Hence they have lazy flight characteristics. In space there is no aerodynamic resistance. Therefore the vehicles veer easily in curves.

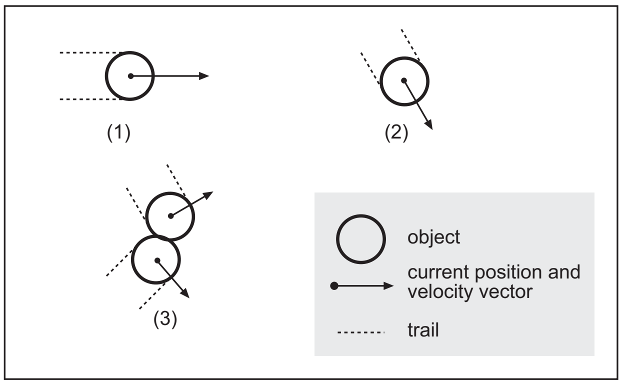

Figure 3: Each object is represented as a circle and has an individual velocity. The two objects in (1) and (2) are moving in different directions. The lengths of the velocity vectors are also dissimilar. In (3) two objects collide and a new velocity is calculated.

2.1.3 Sound effects

The soundtrack in Neon Racer is more than music that supports the game. The soundtrack is created and influenced by the number of vehicles involved in the game and their speed. Each vehicle creates a part of the soundtrack as it moves. A vehicle's sound is loud when it moves quickly and almost inaudible when it is standing still. This way players shape the soundtrack as they play. Additionally the sounds of the moving ships merge with a specially composed background-beat to form an all-embracing composition.

2.1.4 Additional features

Beyond the broad range of existing features, the software design of the game is easily extendable. This makes it possible to add new features, such as bonus-extras or shader networks, quickly. Stability and robustness were also important goals during development. In prior exhibitions the game has run smoothly for a long time without any noticeable drawbacks.

2.2 Table-based Setup

A special rear-projection table-based setup is used for Neon Racer. The compact setup offers enough space to accomodate a projector, a camera and a computer. It also allows us to detect objects on the table's surface, while displaying the virtual game without occlusion caused by objects or players. To gain enough distance for the projector to fill the rear-projection screen and to avoid overheating, a mirror redirects the image. Figures 4 and 5 show the hardware setup of Neon Racer.

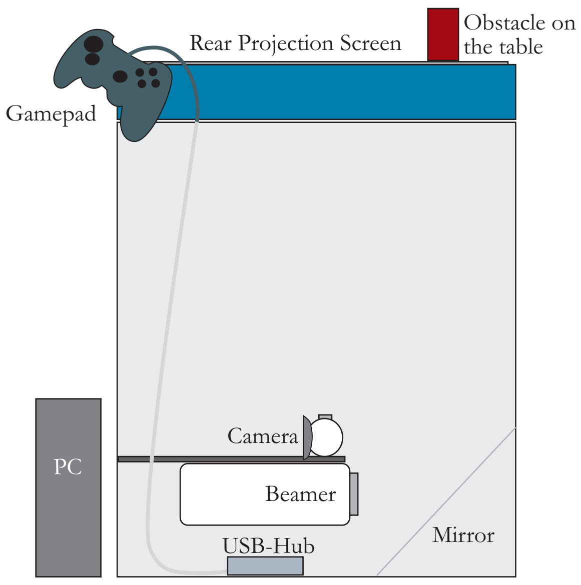

Figure 4: Neon Racer's table-based setup.

A camera situated inside the table records the table surface from below. We use an infrared filter to avoid the camera image being polluted by the projected image. If the environment is too dark for the camera, infrared spots are mounted above the table to provide additional light – invisible to the human eye. Figure 6 shows the resulting camera image.

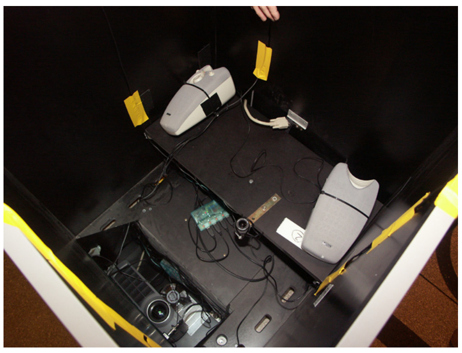

Figure 5: View inside the table.

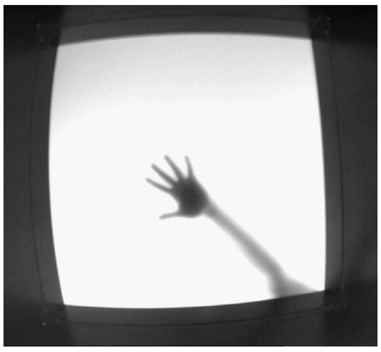

Figure 6: Infrared-image from the camera inside the table.

3 Integrating Real Obstacles

3.1 Camera Calibration

There is an offset between the camera image and the projector image, due to the different positions and focal attributes of the camera and projector. Correct collision detection requires the real world to be aligned exactly with the projected game world.

To eliminate the offset between the two images, a special camera calibration program was created. The basic idea is to find four corresponding points in camera space ((x_1,y_1) to (x_4,y_4)) and game space ((x'_1,y'_1) to (x'_4,y'_4)). The points are used to calculate a projective mapping matrix, which can map the objects' line vertex coordinates (x,y) from camera space to game space (see equation 1). As the projective mapping matrix is a 2D matrix, we introduced a uniform column and row for the z-axis to keep the z-value unchanged.

$$\begin{pmatrix} h'x' \\ h'y' \\ h' \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{1}$$

To calculate the matrix, we need to detect four corresponding point pairs in camera and game space. We thus define four points in game space as our target points and need to find the corresponding camera points. As we cannot see the projected image on the camera image due to the infrared filter, marks are projected on the defined positions in game space. When small objects like coins are placed onto these marks, they are detected and their centre coordinates are saved as corresponding points in camera space. In addition, the corners of the rear-projection screen are also detected on the camera image to extract the borders of the race course.

The original corner points in camera space (x_1, y_1) to (x_4, y_4) are first mapped to a uniform quad. Afterwards, they are mapped to the target points in game space (x'_1, y'_1) to (x'_4, y'_4). This additional step simplifies the calculation of the values a_{11} to a_{32} for the matrix using the eight formulas in equation 2 et sqq.

$$a_{31} = \frac{(x'_1 - x'_2 + x'_3 - x'_4)(y'_4 - y'_3) - (y'_1 - y'_2 + y'_3 - y'_4)(x'_4 - x'_3)}{(x'_2 - x'_3)(y'_4 - y'_3) - (x'_4 - x'_3)(y'_2 - y'_3)} \tag{2}$$

$$a_{32} = \frac{(y'_1 - y'_2 + y'_3 - y'_4)(x'_2 - x'_3) - (x'_1 - x'_2 + x'_3 - x'_4)(y'_2 - y'_3)}{(x'_2 - x'_3)(y'_4 - y'_3) - (x'_4 - x'_3)(y'_2 - y'_3)} \tag{3}$$

$$a_{11} = x'_2 - x'_1 + a_{31}x'_2 \tag{4}$$

$$a_{21} = y'_2 - y'_1 + a_{31}y'_2 \tag{5}$$

$$a_{12} = x'_4 - x'_1 + a_{32}x'_4 \tag{6}$$

$$a_{22} = y'_4 - y'_1 + a_{32}y'_4 \tag{7}$$

$$a_{13} = x'_1 \tag{8}$$

$$a_{23} = y'_1 \tag{9}$$

Spherical distortion of the camera image can lead to misalignment of the real obstacles. Near the border edges of the table, the fisheye lens can lead to distortions of up to 5 mm (see figure 6). In the case of Neon Racer, this can be omitted because the error is usually imperceptible. The vehicles appear to collide accurately with real obstacles.

3.2 Collision Detection

The collision detection in Neon Racer calculates events between virtual vehicles and physical objects, besides handling intersections of virtual entities. To detect physical objects, the obstacle detection system receives a grayscale camera image and first creates a binary image using an adjustable threshold. A contour following filter is applied to the binary image to detect the borders of the objects on the table. The pixel coordinates of the borders are saved but we cannot use the pixel-based collision detection due to performance issues. To accelerate collision detection the objects' pixel borders are simplified by partitioning them into lines. The start and end points for these lines are defined by significant border pixels.

A vehicle's circular shield is used for collision detection, which is also the bounding circle of the vehicle. A collision between a virtual vehicle and a real object can happen anywhere between the current and the future vehicle position. For collision detection with the real objects, several points on the bounding circle are defined and connecting lines between these points on the circle of the current and the future position of the vehicle are created (see figure 7). Each vehicle line is tested to determine whether it is completely or partly inside an object's axis aligned bounding box. If this is the case, line-line intersection is used to test for an intersection with each of the object's border lines. When an intersection occurs, the normal vector of the object's border line is used to calculate a correct collision response. If there is more than one intersection, the intersection closest to the centre of the bounding circle of the vehicle's current position is chosen as the final collision point. Figure 8 shows an example program representing the vehicle as its bounding circle and the real object as border edges.

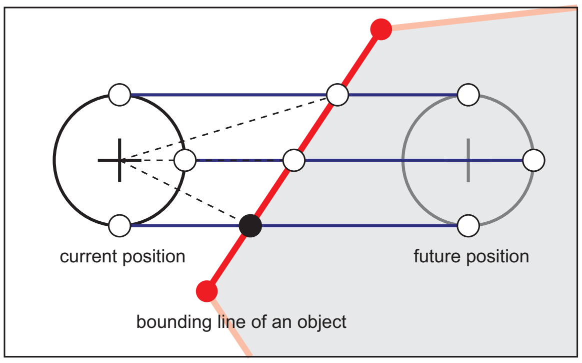

Figure 7: A vehicle, represented by its bounding circle, colliding with a real object, represented as border edges. The lines connecting the vehicle's current and future position are used to determine the point of collision. In this case the result is the closest point, shown in black.

Line-based collision detection works faster than pixel-based collision detection. Especially when the vehicles move fast and thus make bigger steps the pixel-based collision detection needs to check a huge number of pixels, in contrast to the line-based collision detection whose complexity is independent from the vehicles' step width. On the other hand the pixel-based collision detection is more precise and we could discriminate against the vehicle being "inside" or "outside" the real object. In favour of calculation time we disregard impreciseness resulting from the simplification. At the four border edges of the table we additionally use circle-line intersection, to avoid vehicles getting stuck at the border of the race course. However, in contrast to exclusive line-line intersection this would consume too much calculation time to check each object's border for a collision.

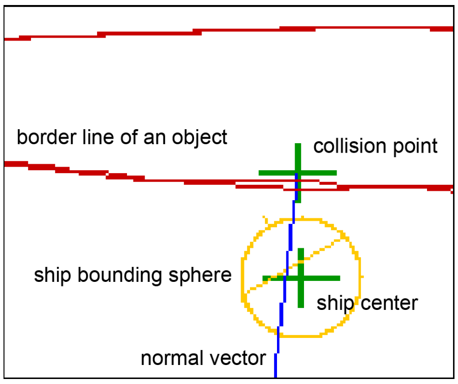

Figure 8: The collision between a virtual vehicle and a real object in an example program. The crosses mark the vehicle's centre and the collision point; the vertical line represents the normal vector of the object's border at the collision point.

4 GPU-Particles

4.1 Motivation and overview

To enhance the speed feeling for the player, we decided to render motion lines behind the moving vehicles. As our application includes a number of computer vision tasks that are done on the CPU, we decided to compute the motion lines on the GPU. Actually the motion lines are done using a simple particle system. An example for rendering particles on the GPU may be found in [4].

Our particle system is able to form motion lines behind the moving vehicles. Particle emitters are located at the vehicles' positions and move with them. Each frame, particles are emitted at the vehicle's current position. The particles stay where they have been emitted until their death. During their lifespan they only change their alpha value. After the particle has died, it is emitted again at the current position of its emitter.

4.2 The particle algorithm

The particle system requires knowledge of the position and age of each particle and should be calculated entirely on the GPU. Using a fragment shader it is possible to write floating-point values (e.g. xyz-coordinates) into a texture. The OpenGL extensions for vertex and pixel buffer objects offer ideal support for this method because vertex data saved in an array can be interpreted as rgb-information of a texture (see [2] and [3]). Conversely, it is possible to read an rgb-value as an xyz-vertex position. To maintain the accuracy of the vertex position, it is necessary to save the rgb-value as floating points. This is achieved using floating point textures.

The whole algorithm to display the particles is divided into three render passes:

1. **Creating the emitterPositionTexture.**
   Render the emitters' current positions into the texture `emitterPositionTexture`.
2. **Computing the particles.**
   Render the particles and increase their age. If necessary, compute new particle position using the `emitterPositionTexture`.
3. **Rendering visible output.**
   Render the particles as point sprites.

There are three shaders, one for each render pass. The shader which calculates the next render pass uses the output from the previous render pass. The output of render passes one and two is written into PBuffers. These buffers have the advantage that they can be bound as textures. This is more efficient than copying the result of the render pass from the framebuffer into a texture. The resulting textures are used as input for the following pass. In addition the usage of PBuffers is crucial because the framebuffer cannot store floating point values.

The particle system has a fixed number of particles which does not change during runtime. Changing the number of particles is not necessary in the application, but with a few modifications this would also be possible. Each emitter has the same number of particles. Originally each moving object had its own particle system. As there are only four vehicles in Neon Racer, we made some modifications to use only one particle system for all four vehicles.

4.2.1 Creating the emitterPositionTexture

To be able to retrieve each emitter's position in a fragment shader, the position must be rendered into a texture (see figure 9). The rgb-values of the texture can be referred to as the xyz-values of the emitters.

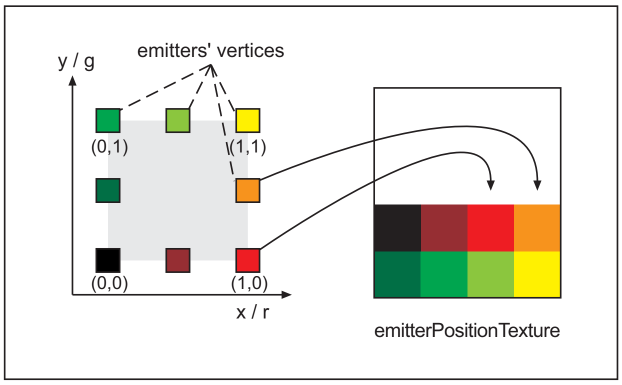

Figure 9: On the left side, a quad with eight emitter vertices is shown. During the first render pass the positions of the eight emitters are stored into a texture like the one on the right.

4.2.2 Computing the particles

After updating the particle emitters' positions, the particles' new positions and ages are computed. Properties of the particles are stored in three vertex buffer objects. The first vertex buffer object stores position and age. It is also defined as a pixel buffer object. This makes it possible for rgb-values of a PBuffer to be copied into the buffer object and then interpreted as xyz-values. In the second vertex buffer object, the particles' color are saved. The emitter determines a particle's color. The third vertex buffer object contains the texture coordinates to read from the `emitterPositionTexture`. These coordinates determine which emitter a particle belongs to.

A shader cannot read and write the same texture simultaneously, thus there are two floating point textures that save the previous and current positions and ages of all particles. Figure 10 shows an example of a texture containing the particle positions and their ages. A particle's age is encoded in the texture's alpha-channel.

Particles who have died are emitted again in an infinite loop. In the shader, the previous position and age of the particle is retrieved. Each frame, the age is decreased. If the result is smaller or equal to 0, the particle is dead and has to be emitted again. To do this, the age is reset to 1 and the particle's new position is set to the emitters' current position. This position is fetched from the `emitterPositionTexture` using the texture coordinate that defines which emitter the particle belongs to. The retrieved value is then set as the particle's new position (see cg code in 11).

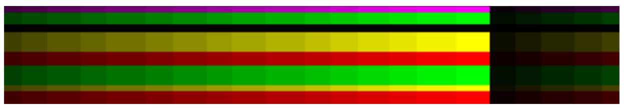

Figure 10: This floating point texture stores the positions and ages of all particles. The position is saved in the rgb-channel and the age in the alpha-channel.

```
// get position and age of last frame
outcolor = texRECT(prevPosition, texCoord);
// reduce life
outcolor.a -= deltaLife;

if (outcolor.a <= 0.0) {  // rebirth of particle
  // get new position and set full life
  // (saved in alpha value of
  // emitterPositions texture)
  outcolor =
        texRECT(emitterPositions,
          texRECT(emitterNumber, texCoord).xy);
}
```

Figure 11: The cg code for updating the emitter's position and age.

4.2.3 Render visible output

In the final render pass, the updated particle system is rendered. The current particle positions are read from the texture as vertex data and used to draw the particles. To render the particles efficiently, they are displayed as sprites, using the `GL_ARB_point_sprite` extension (see figure 12).

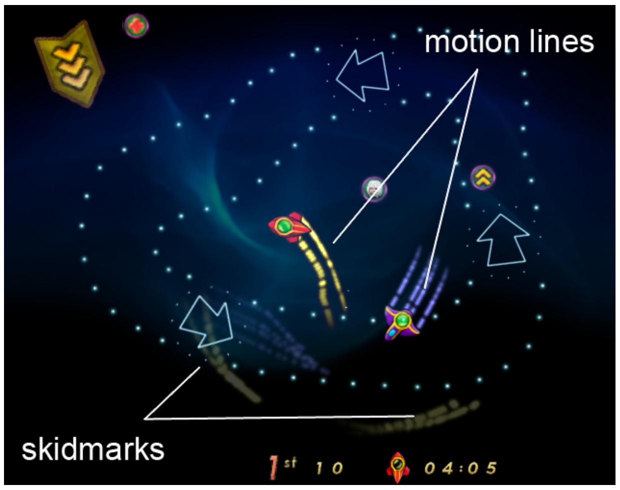

Figure 12: Motion lines and skidmarks made with the particle system. When speed-up items are activated, the particles are baked to leave skidmarks on the ground.

4.3 Skidmarks

Vehicles leave skidmarks on the ground if they use a speed-up item. A skidmark is a trail of particles which stays on the ground until the end of the game (see figure 12). This is accomplished by rendering a snapshot of the particles emitted by the vehicle into a PBuffer. The snapshot is then blended over the background-texture.

4.4 Results

On an nVidia GeForce6800, the algorithm renders 8192 particles at 50 fps. With more particles the CPU usage increases. This could be fixed by using the `GL_ARB_point_parameters` extension. Rendering only points, without using the `GL_ARB_point_sprites` extension, there were approximately 65,000 particles at 60 fps.

It would be possible to change the position of the particles and give them velocity, by including an additional render pass (or multiple render targets). Special effects such as explosions could be achieved this way.

5 Summary

In this paper we have examined possibilities for future AR entertainment applications and described a sample application. Neon Racer offers important features for entertainment installations, such as technical robustness, a user-friendly interface and engaging gameplay (see figure 13). Neon Racer uses a novel approach for controlling the game by incorporating both physical and "digital" interaction. Each player and spectator can use real objects to influence the race, while players steer the vehicles with traditional gamepads. This combines a widely familiar computer game control with an intuitive physical interface. Both direct physical contact and gamepad participation are necessary to be successful. Neon Racer enhances not only the classical racing game, but creates a setup which allows for direct physical interaction with the game world.

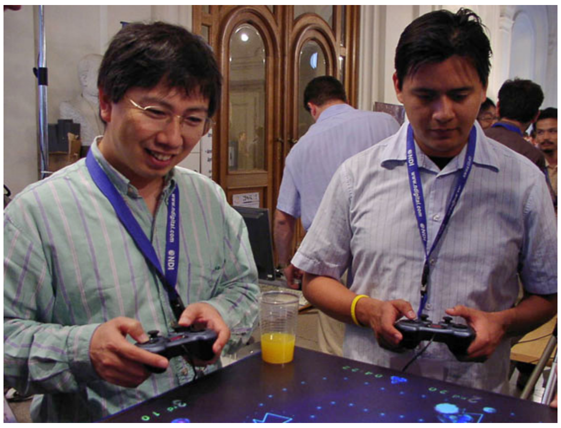

Figure 13: People playing Neon Racer at ISMAR 05 in Vienna.

The technical setup is an all-in-one table which is fairly easy to configure without major adjustments. The game's technology is conveniently built into the table, with the effect that users more openly engage the game without focusing on the technical setup. Through the use of infra-red image processing Neon Racer is employable in different lighting conditions. Locations can be both bright or dark. The game setup is flexible and can be placed in bars, exhibitions, shopping malls etc.

Neon Racer was demonstrated successfully at the Pixelspaces Exhibition 05, part of the Ars Electronica festival in Linz, the ISMAR 05 and the EUROPRIX Multimedia Top Talent festival 05 in Vienna (see figure 14). Even after intense play, no drawbacks in performance occurred. We were amazed by the consistently positive feedback of the audience. Neon Racer appears to have bridged a gap between entertainment and technological research.

People of all ages enjoyed the game. We observed that handling of gamepads needs some practise. Players not used to them had difficulties to control their vehicles. Advanced users are interacting more with the objects on the table to influence the game. If someone started to move objects around other players and spectators also joined in. People were amazed that objects they put on the table are integrated into the game.

Future considerations for Neon Racer focus mostly on simplifying the vehicle's control. Graphical enhancements with custom shaders and physical models can improve the pace and make Neon Racer more challenging. Integrating different levels may tempt to replay the game more often.

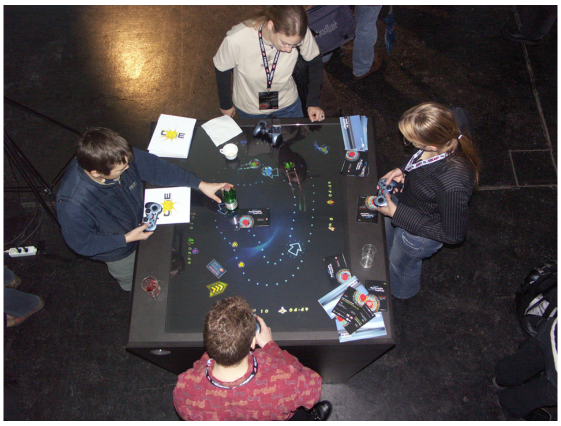

Figure 14: The Neon Racer table at the Top Talent Festival 05 seen from above.

6 Acknowledgements

The authors would like to express their gratitude to Mandala Weber of Circled Cube for the sound design and Horst Hörtner of the Ars Electronica Futurelab for supplying hardware for the first table-based setup. We would also like to thank Thomas Weilguny for additional graphics and Jürgen Zauner and Peter Brandl for their great dedication. For help with organisation we thank Michael Haller.

References

[1] Steve Benford, Carsten Magerkurth, and Peter Ljungstrand. Bridging the physical and digital in pervasive gaming. *Commun. ACM*, 48(3):54–57, 2005.

[2] NVIDIA Corporation. Using vertex buffer objects (vbos), 2003.

[3] Silicon Graphics. Opengl extension registry. URL, http://oss.sgi.com/projects/ogl-sample/registry/, 2003.

[4] Lutz Latta. Building a million particle system. URL, http://www.2ld.de/gdc2004/MegaParticlesPaper.pdf, 2004.

[5] Carsten Magerkurth, Timo Engelke, and Maral Memisoglu. Augmenting the virtual domain with physical and social elements: towards a paradigm shift in computer entertainment technology. *Comput. Entertain.*, 2(4):12–12, 2004.

[6] Wayne Piekarski and Bruce Thomas. Arquake: the outdoor augmented reality gaming system. *Commun. ACM*, 45(1):36–38, 2002.