

# off BEAT

Peter Messenger, 0327186, messenger.mine@gmail.com

Matthias Muehlich, 0308771, mmuehlich@gmx.li



## An outer space audio game

---

### Short description

Off Beat is an outer space audio game where the player flies along a path through a field of obstacles and tries to hit nodes that appear in the rhythm of the music. Each node that was hit on time explodes and rewards the player with points according his timing, missed objects reduce the high-score.

### Short walk-through:

The game is started by pressing "P" (play / pause), immediately the player is set on the track and the music starts to play. Navigation is done by the computer, the only task is to click on the stars / spheres that appear when they are on-beat (indicated by green color). Targets that were hit too early cannot be hit again and will count as missed objects.

### Controls:

For the normal game-play, only the mouse is needed and "P" will start / pause the game.

Other controls are:

- Light position:
  - F move light left
  - H move light right
  - T move light away (z-direction)
  - G move light to camera (z-direction)
  - N move light down
  - Y move light up
- Cameras:
  - C changes the current camera:
    - three cameras are available, the level can be played using all three cameras however camera positions within the levels are fixed. When pause is set, the cameras can be moved by
      - A-S-D-W and E-X (up down, only on some cameras)
      - mouse movements (only on some cameras)
      - each one of the three cameras will remember its position, when switching from

camera to camera the old position will be reused

- Debug functions:
  - 9 saves the whole scene to a file – is disabled to avoid overwriting the existing file
- Rendering:
  - F1 displays some help information
  - F2 toggles frame-rate information
  - F3 switches wireframe on or off
  - F4 switches between bilinear texture filtering and nearest neighbor interpolation
  - F5 mip-mapping: no mip-mapping, nearest neighbor mip-mapping and linear interpolated mip-mapping
  - F7 lock framerate on 60 fps
  - F8 view frustum culling:
    - no view frustum culling
    - culling of the level geometry (the boxes) only on near and far Z
    - culling of the level geometry against the view frustum
    - culling of the level geometry and the objects of the scenegraph against near and far Z
    - culling of the level geometry and the objects of the scenegraph against the view frustum
  - F9 transparency on / off
- Turn features on / off
  - 0 turn off mouse picking
  - 1 turn off main rendering
  - 2 turn off lightshafts
  - 3 turn off bloom
  - 4 turn off environment mapping (currently disabled)
  - 5 turn off lens flares
  - 6 hide the crosshair cursor
  - 7 turn off particle systems

## Objects and animated objects

The current scene still contains some debug objects that can easily be exchanged for more complex models. However, the simple shapes of the objects create a very special atmosphere, therefore these were not changed yet.

All objects along the level are environment mapped, the targets itself are key frame animated.

## Visibility and view frustum culling

The scene contains three types of geometry: Any number of objects can be loaded using an xml-like scene description file and targets are placed along the camera path according the beats of the music. Both groups of objects are controlled by a scenegraph which takes care of updates, drawing and transformation order.

The geometry along the camera path consists of objects that are not contained in the scenegraph but are instances of one single object that is moved and scaled. This approach was chosen because drawing the whole scene geometry using the scenegraph resulted in very bad performance. This performance could not even be reduced by view frustum culling on the scenegraph.

Therefore different types of view frustum culling are implemented:

- Objects of the level geometry are automatically removed when the player has passed them by a certain distance. Since the player will on a spline along the z-axis this is no problem for the normal game, but may be disturbing when the debug camera is used to turn around and observe the level behind the player. However, this culling presents a large performance gain and does not disturb the normal gameplay.
- Additionally, the level geometry can be culled against the near and far Z-plane. If this culling is enabled, the objects behind the camera are first culled and after some distance deleted as above. Since speed is more important than exact culling, the objects centers are used for culling, not the bounding boxes or spheres, using the bounding spheres gave no performance gain but some additional overhead in calculating the bounding spheres and testing them against the frustum.
- The geometry can also be culled against the whole frustum. However, since this increases performance only minimal, the culling is implemented as demo culling and the culling planes are set slightly smaller than the view frustum, so the culling becomes visible. Again, this culling uses object centers since the bounding-sphere / box calculation did not bring any performance improvements.
- Objects of the scenegraph can be culled against the near and far Z plane and the whole frustum just as the level geometry. In the scenegraph, objects can be grouped to be culled together.

## Transparency

Blending is a central part of the project. Almost all textures, especially those used for particle effects and similar, contain 0 alpha values. Between renders, scenes are blended using either `glBlendFunc(GL_SRC_ALPHA, GL_ONE);` (particle systems) or `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)` (normal blending). For some particle systems depth testing was disabled in lieu of performing a costly sorting function.

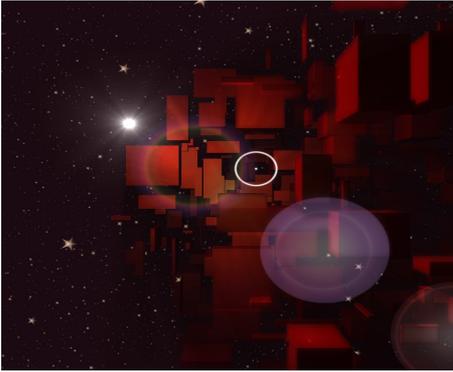
## Effects

The following effects were implemented:

### Environment mapping:

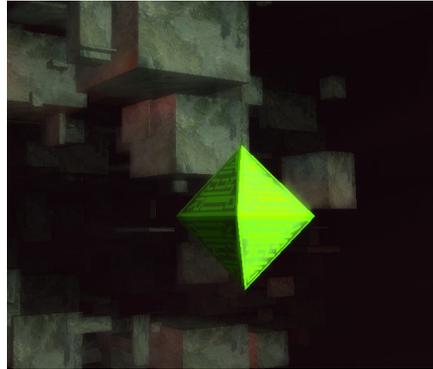
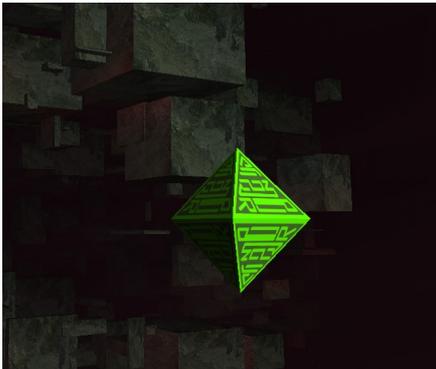


Simple Cubemapping was implemented. A static cube map texture is used (not generated on the fly). The Cube map texture is loaded onto the graphics card, then bound before an object with environment mapping enabled is rendered. In our game Demo it is used on the track cubes. It is implemented very discretely (blend factor over existing texture 0.5) . Implementation was partly based on the tutorial found at ozone3d.net. [http://www.ozone3d.net/tutorials/gls\\_texturing\\_p04.php](http://www.ozone3d.net/tutorials/gls_texturing_p04.php)



### **lens flares:**

The lens flare effect is implemented using the geometry shader. Similar to our particle systems, quads are generated and the selected shapes are taken from a sprite sheet. In this case three textures were used to generate the effect. Positions are calculated in screen space, using the light position and the center of the screen.



### **Bloom:**

The bloom effect is a postprocessing effect. The lighted scene is rendered into a scaled down viewport framebuffer (factor 8). The downscaled image is then gaussian blurred with a 5x5 kernel. The blurred framebuffer image is then drawn on a 1x1 plane in the full viewport resolution and then blended over the full render of the scene. This effect was mainly based on the nVida GPU gems description of the glow effect.

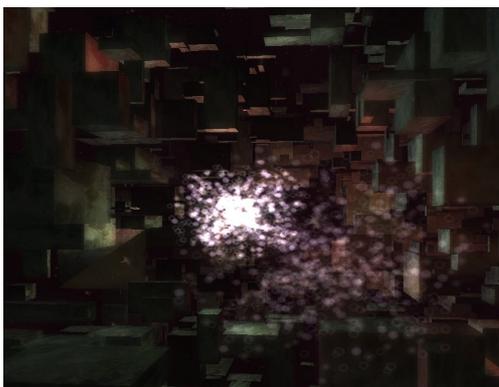


## **lightshafts:**

Lightshafts is the most performance intensive effect in this project. The scene is rendered at half resolution with a minimal shader. All objects are rendered black except the light, represented by a sphere, which is rendered white. Every pixel is sampled towards the light source, adding weighted color (white or black) along the way. The result are lines which are brighter the closer they are to the white light source. The sampling iterations are very costly and were set at 70 per pixel for best performance/quality.

This effect was based on the GPU gems article on light scattering as well as the implementation of Fabien Sanglard.

<http://www.fabiensanglard.net/lightScattering/index.php>



## **particle systems:**

Three types of particle systems were implemented, of which only 2 are visible in the final game demo. All particle systems are implemented entirely on the GPU, which provides for a huge speed

boost but makes them less flexible.

The first system is the explosion particle system. All systems can be bound to other objects and will then take over their position. The particles' positions are generated randomly along with a few properties, most significantly the velocity property which provides the vector along which a particle will travel. On the GPU each particle is then moved along this vector based on the update time. The directions are randomly generated then normalized vectors, meaning the particles expand in a spherical behavior.

The second type of particle system expands on the first by also allowing for animated particles. The sprite sheet is traversed according to a timing scheme and is looped. This provides an interesting effect. This system was not implemented, mainly because we could not acquire a proper animation and had no time to generate one ourselves.

The third system is based on description of "imposters" in the "Real-time Rendering" (Akenine-Moeller et al. ). Basically, an object is rendered in real time to a frame-buffer, which is then used to texture each particle. This object is lit with the same light source as the scene, but the light uses a fixed z-value. This means, the object can be animated and lit by the same light source as the rest of the scene once, but then can be displayed many (10'000 times in our case) at a fraction of the cost it would use to render each individual object. The effect is interesting and could be optimized using more objects in smaller adjacent cubical cells.

## Some Implementation details

This section describes some implementation details.

### **scene description:**

Although the current scene is still a testscene (which will be replaced by a more complex scene within the next week), the surrounding scene can be loaded and saved into a xml-like file which allows easier modification and storage of a scene. A future version of the game could easily implement a scene editor which generates these files so players can create their own scene and share them. The scene file represents both, the structure of the scenegraph as well as the structure of the objects, some example files can be found under /Scenes/.

### **scenegraph:**

The scene description is used to create a scenegraph which updates and draws itself. The scenegraph is also capable of handling view frustum culling. This was explained in great detail in the first hand in documentation.

### **object libraries:**

VBOs, textures, shaders and other resources are stored in libraries where they can be queried easily.

### **level object generation:**

The level objects are generated randomly along the track which is defined by a spline. The Basic idea is similar to a particle system. The Level uses a constant amount of elements (1000) which it adjust along the constant z-axis. This method of culling proved to be by far the most efficient way of rendering the level. This method of level generation also results in a compact looking track without the necessity of having to model the whole track by hand. Additionally, this method would allow for quick adjustment to another theme song, as the length of the track is directly based on the length of the song. The individual control points for the catmull rom spline are set by hand (30 control points).

## **libraries / Tools:**

- fmod (sound)
- glew (gl extensions)
- glfw (gl window management)
- glm (math library)

All objects were created either manually or using blender.

