Parallel Multi–View Rendering on Multi–Core Processor Systems

Jan Ohlenburg[*]
Fraunhofer FIT
Germany

Wolfgang Broll[†]
Fraunhofer FIT
Germany

1 Introduction

Dual–core processors have now become a standard in modern desktop and notebook computers and it is only the beginning of a new trend leading to massive multi–core processor systems. In order to utilize multi–core systems applications have to be highly multi–threaded. Multi–threaded rendering is still a non trivial process, since update and rendering processes have to be synchronized very carefully. Failure to synchronize correctly will result in a drastic performance loss.

We present an approach to multi–threaded rendering taking advantage of the power of today's and future multi–core systems using OpenGL. This approach supports a high number of parallel rendering threads, simply limited by the capabilities of the computer, where each rendering thread may have its own camera, render state and refresh rate while sharing resources like textures and display lists. In order to achieve the maximum performance one thread – called the manager– performs the updates of the scene graph and blocking all render threads during this time period.

2 Implementation

The first key issue to maximize performance in a multi–threaded rendering system is to avoid context switches, i.e. each rendering thread has to have its own context and may only operate on this context. Therefore, calls to switch the OpenGL context can be eliminated completely. The second key issue is the efficient synchronization between the update process and the rendering processes. During the update process changes to the scene graph are processed, resulting in renewed display list, vertex buffer, textures and so on. Since these resources are shared among all rendering threads, the update process has to be seen as a critical section and no rendering thread may be active while the update of the scene graph is done.

The straight forward way to achieve multi–threaded rendering is to perform an update, start all rendering processes, wait for the completion of all of them, then do the update for the next frame. While this approach is correct, it has the limitation that each rendering thread will have the same refresh rate. In case the application has a main view and a small top view in the corner of the screen, it would be desirable to have a much smaller refresh rate of the top view in order to have a higher refresh rate in the main view.

Our approach allows us to define an individual desired refresh rate for each view. The approach requires each render thread to have a mutex, which is locked during the render process, and two conditions, one for the manager and one for the refresh rate. A condition is a synchronization primitive used to let a thread wait until a specific condition is met or a specific time has expired. The rendering thread signals the update thread when desiring an update, waits for the next update, then locks its own mutex, renders the scene, and finally conditionally waits for the next frame (see Listing 1).

```
while (!isTerminated())
  mutex.lock();
  manager->notify();
  managerCondition.wait(lock);
  render();
  condition.timed_wait(lock, nextFrameTime);
  mutex.unlock();
```

Listing 1: The render thread loop

The update thread on the other hand, waits for the notification of one of the rendering threads and then locking the mutexes of all of them. This ensures that each render process can finish before the next update cycle will start. After acquiring all mutexes, the update is performed, followed by a release of all mutexes and a notification to all waiting rendering threads (see Listing 2).

```
while (!isTerminated())
  mutex.lock();
  condition.wait(lock);
  for each render process
    renderer->mutex.lock();
  update();
  for each render process
    renderer->mutex.unlock();
    renderer->notify();
  mutex.unlock();
```

Listing 2: The update thread loop

Depending on the update time for a new frame, the chance that a render thread signals for an update while the update is currently performed raises with the update duration. In this case the same update can be used for all waiting render threads, saving valuable time. On the other hand, if the update can be performed very quickly, updates are less likely to be shared, but since the update is affordable it will not lead to any significant decrease of the frame rate.

3 Conclusion

Our presented approach provides a general solution for rendering a single scene into multiple views in parallel taking advantage of current multi–core processor systems, where each view may have its own camera perspective, render state and refresh rate. The synchronization procedure between the managing update thread and the rendering threads allows for optimal update rates. Although this solution currently cannot applied to Direct3D, since multiple swap chains cannot be rendered in parallel, we expect this to change in upcoming Direct3D versions.

[*]e-mail: jan.ohlenburg@fit.fraunhofer.de
[†]e-mail:wolfgang.broll@fit.fraunhofer.de