# 2
# Parallel Game Engine Architectures

This chapter presents different forms of organizing the computation of a game on a multicore system. Most of the architectures presented in this chapter can be found in the work by Monkkonen [Monkkonen06] and Dawson [Dawson06]. We use a simplified set of game engine tasks to make easier the comparison between the different architectures. This simplified set does not consider tasks that are related to streaming, audio and networking.

## 2.1 A Single Threaded Game Loop

Current mainstream game consoles, Xbox 360 and Playstation 3, are both multicore systems allowing parallel computing to be performed. In order to make proper use of such systems, it is necessary to go beyond the simplicity of the single thread main game loop. An example of such simple main game loop is presented in figure 2.1.

The Culling & Sorting task executes algorithms for scene graphics culling and resource sorting. Scene graphics culling removes parts of the virtual environment not seen by the camera. Resource sorting optimizes rendering by allowing reduced resource changes. The Rendering task renders the scene by configuring the render device and making render calls. The Input task detects the input devices states. The AI & Scripting task execute AI algorithms (such as path finding) and general scripts. Scripts may be used for implementing object behaviour and special procedural sequences. The Animation task animates the objects of the game based on an animation sequence. The Physics & Collision Detection task executes the physics simulation and also the collision detection of a game. The Present Back Buffer task simply shows the contents of the back buffer where the scene has been rendered. This task is at the end of the loop in order to increase the parallelism between the CPU and the GPU. By processing others tasks on the CPU before presenting the back buffer, time is given to the GPU to finish the rendering of the scene so that when this task is reached, no stall is necessary [DirectXDocumentation].

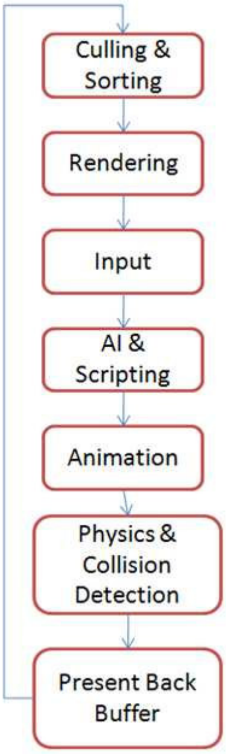

Figure 2.1: A Single Thread Game Loop

## 2.2 Synchronous Parallel Function Architecture

This architecture takes the tasks from the single threaded game loop presented in figure 2.1 and sets the ones that can run in parallel in different processors, as shown in figure 2.2. The Animation and Physics & Collision Detection tasks are dependent on the Input and AI & Scripting tasks. The benefit of this architecture is that it is easy to incorporate legacy single threaded technology into this architecture, making it an inexpensive technology upgrade. The drawback is that this is not a scalable solution. Generally, there is a limit to the amount of processor usage that can be achieved by dividing the game technology tasks in this way. So, this architecture is a fine solution for the current generation of multicore processors, such as the Intel Core 2 CPU family and the Xbox 360. However, CPUs with a higher amount of cores will probably not be used in an optimal way.

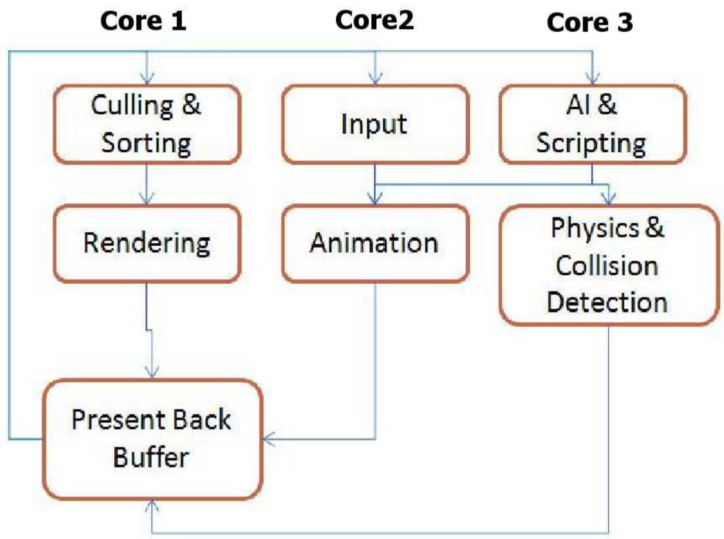

Figure 2.2: A Synchronous Parallel Function Architecture running on 3 cores

## 2.3 Asynchronous Parallel Function Architecture

The asynchronous parallel function architecture also divides the tasks of the main game loop between the multiple cores. The difference is that the tasks on each core are allowed to run without needing to wait for tasks on other cores to finish (figure 2.3).

In this architecture, each task uses the latest data available in order to make its own processing. For example, the Culling & Sorting task gets the latest data on the scene objects, their position, orientation, etc, even if the Physics & Collision Detection is changing the scene objects information. This could be accomplished by having both tasks accessing the same data in a synchronized way. An alternative solution is to make the Culling & Sorting task use a buffer to read the data with the latest scene object information, while the Physics & Collision Detection task writes on a different buffer. When finished, the buffers are swapped and the Culling & Sorting starts reading from the new scene data buffer.

This architecture shares the same benefit with the synchronous approach; That is: the ease of implementation for current single thread technologies. Another benefit is that, since the tasks are mostly asynchronous, information will be more up to date as time passes. For example, if the Graphics tasks (Culling & Sorting, Rendering, Present Back Buffer) take a long time, the AI

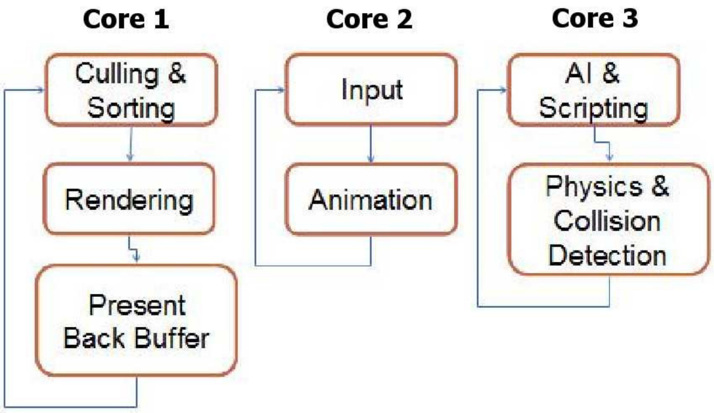

Figure 2.3: An Asynchronous Parallel Function Architecture running on 3 cores

and the Physics Tasks could run another time and give a more up-to-date information for the next frame based on the previous time step.

This architecture also shares the drawback with the synchronous approach of not scaling well for a higher amounts of cores. Another drawback is the wasting of processor resources. When a task executes a second time on the same frame and uses the same input with nothing being changed from the previous processing, the task is making bad use of processor resources. For example: a scripted AI for an enemy tests visibility of the player a second time and sets the enemy to a combat state again. The resources could be allocated for additional processing of another task, such as physics, that would allow a more realistic physical simulation on the game.

## 2.4 Data Parallel Architecture

In the data parallel architecture, instead of dividing the processing based on the tasks, the processing is divided based on the data and the same tasks execute on each processor on different parts of the data. For example, the scene objects could be equally divided between the processors as shown on figure 2.4. Each thread in figure 2.4 would take an equal amount of scene objects to process.

The main benefit of this architecture is the scalability, as it scales well for any number of processors. The drawback is that not all game technology will be easily or efficiently implemented this way. For example, if a game engine uses an octree to process culling in a game, it is not simple to divide the culling process of the scene objects among cores. The problem is that this architecture, as the previous ones, is still based on executing single threaded algorithms, a

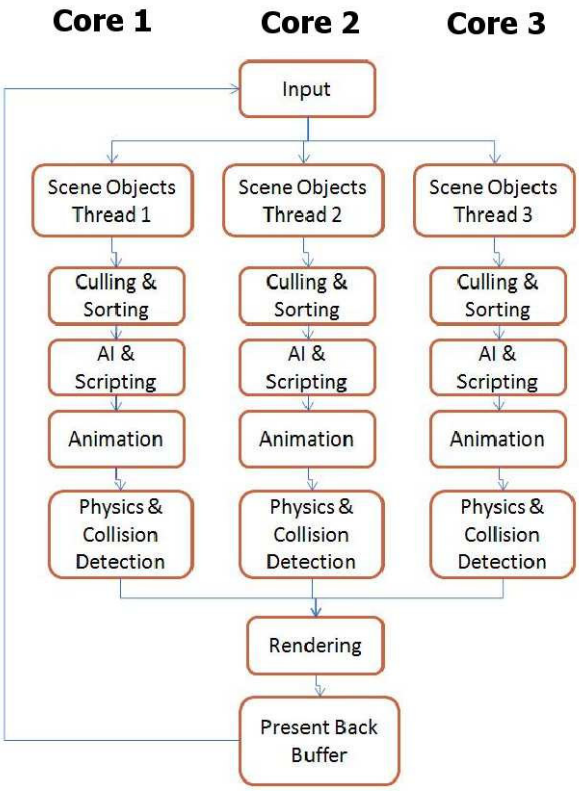

Figure 2.4: A Data Parallel Architecture running on 3 cores

feature that brings difficulties when we try to implement game technologies on multicore systems.

## 2.5 Pipeline Architecture

In the pipeline architecture, each core takes a specific task from the main game loop to process and, after finishing its computation, passes the results to the next core in the pipeline. After delivering the results, the core starts computing the same task, but for the next frame, as presented in figure 2.5.

This architecture can offer a good frame rate being only limited by the task with the longest time to be completed. This task is the bottleneck of the pipeline and optimizing it will increase the frame rate.

The main drawback of this architecture is that the input results take a long time to be displayed on the screen. This is because the computation of each particular frame still takes a long time to be completed. As computers increase in number of processors and the frame computation time increases, this architecture will probably become a poor solution for a game engine architecture.

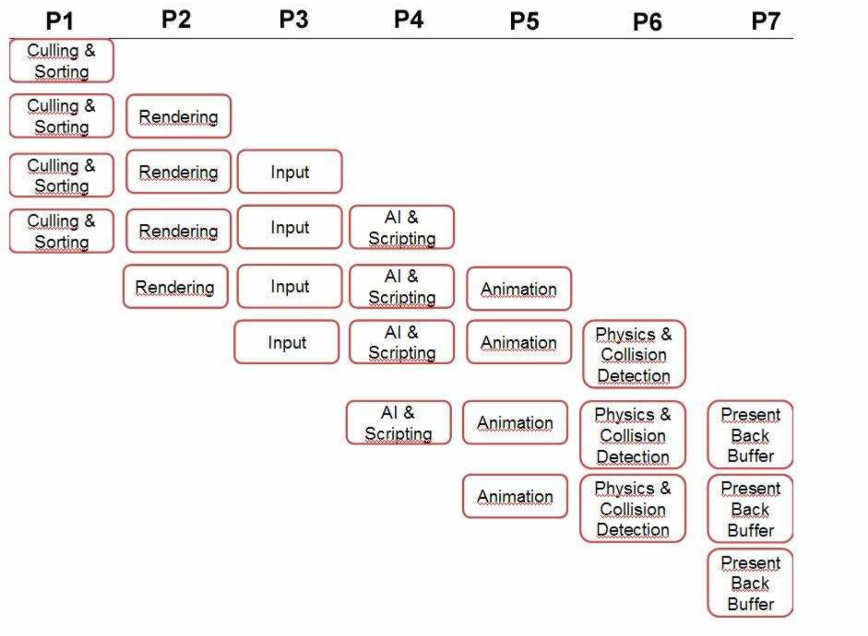

Figure 2.5: A Pipeline Architecture

## 2.6 Fully Parallel Architecture

Whenever possible, the fully parallel architecture makes use of fully parallel algorithms for processing the main game loop tasks. For example, this architecture can use a parallel octree search algorithm for culling processing. It also mixes ideas from the parallel function and data parallel architectures in order to maximize scalability. Figure 2.6 shows this architecture.

The amount of tasks on each core is generally different, mainly because some tasks are single core in nature (*e.g.* rendering and input). Therefore, the technology underlying the fully parallel architecture should have a complete control over the process of how the workload of a task will be distributed among the cores. For example, the first column in figure 2.6 has less tasks to process than the third column. The third column has a lesser amount of tasks to process than the second column, but the Rendering task usually takes much more time to process than the Input task. Because of this task organization, it would be interesting if we could give to the core responsible for processing the first column, a greater amount of work for the AI & Scripting task than the amount given to the other cores. For example, we could divide the AI & Scripting workload in 60/30/10 percent of work on each core. So the core responsible for the first column takes 60% of the task to process, the core responsible for the second column takes 30%, and the final core takes 10%. This would allow a more balanced workload among each core.

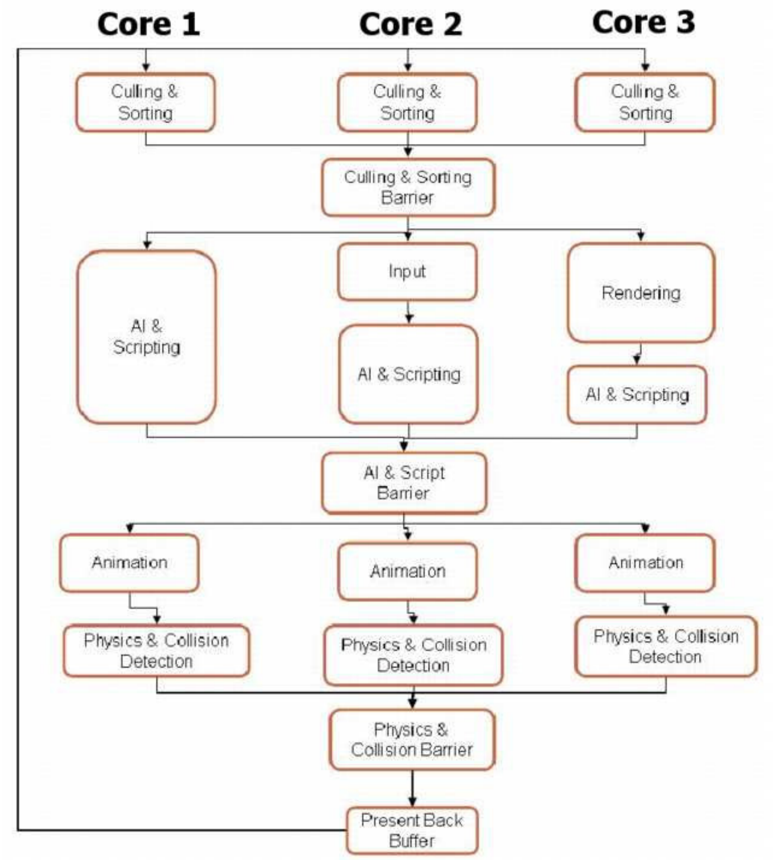

Figure 2.6: A Fully Parallel Architecture running on 3 cores

Some barriers are added to the architecture because some tasks need synchronization. One example is the Rendering task and the Culling & Sorting Tasks. Only after the culling and resource sorting tasks are finished that the rendering can proceed. So the Rendering task only executes after all cores reach the Culling & Sorting Barrier.

As in the case of the Data Parallel Architecture, the main benefit of this architecture is the scalability. However, in comparison to the data parallel architecture, the fully parallel architecture has the advantage of having no obligation of developing a technology based on scene objects. The drawback is that the technology used for implementing this architecture is considerably different from any legacy single threaded technology, what makes it more expensive to develop.

## 2.7 Some Final Conclusions

The next generation of processors and game consoles will come with more cores to handle the execution of the game tasks than those found in current machines. Therefore, it is important to use scalable architectures, in order to have optimal use of the processor resources. In this aspect, the Data Parallel Architecture and the Fully Parallel Architecture are good options.

However, we recommend the fully parallel architecture, because it is the only architecure that fully embraces parallel execution by using parallel versions of classic algorithms (such as octree culling). The fully parallel architecture is also flexible in the sense that it allows parallel execution of scene objects as in the data parallel architecture.