Abstract

This article describes a language and framework that aims to ease soft real-time and concurrent programming for games. The defined language extends the standard C 99 language, adding some new constructions to it. Its standard library provides some graphical utilities for game development, object file loading, rendering optimization passes, built-in physical simulation, high-level collision detection, 3D audio and shader loading. It also provides basic concurrent programming structures and real-time support by some functions which changes real-time exception handling. This extension, CEx, was implemented as a extension of the Clang compiler that generates code for the Low Level Virtual Machine (LLVM). From this implementation, it was possible to provide an environment for game development, which comprehends the language, and an intermediary library for concurrency, real-time signals and graphics.

Keywords: Parallelism, Gaming Framework, Concurrent Languages, Soft Real-time

Author’s Contact: {dedebs,fbafelipe,olinto}@inf.ufsc.br

1 Introduction

There is a growing demand for tools which provide better abstractions, debugging support and productivity, for systems programming. Thus, this is not different with real-time and concurrent applications. These programming tools are commonly added as a library to the programming language. This is truth for C and C++. Other languages as Erlang [Armstrong 2003] and Concurrent C [Gehani and Roome 1989] and C++ [Gehani and Roome 1988] have specific language instructions which promote the same or equivalent capabilities as those libraries. Hybrid solutions provide both instructions and library support. In this last class is the CEx language, which provides concurrent and real-time instructions while focusing on game development.

Games are real-time applications and they tend to use lots of concurrency nowadays [Bethke 2003; Eberly 2004]. They use graphical engines, which are known to implement the three greater steps of a game: physics with collision, artificial intelligence and rendering. These are performed up to 60 times per second [Akenine-Möller et al. 2008; Eberly 2004]. The growing complexity, mainly from games considered as state of art, implies in the need of tools, techniques, libraries and languages to make the software development swifter and robust. Then, most of the enterprise solutions involve commercial engines, and as a consequence some language that the engine supports. These commercial engines provide parallel programs and abstraction, however they usually don’t do this in languages which exposes the code parallelism with instructions.

The CEx language uses instructions only for the concurrent and real-time parts. The engine parts are all provided as libraries, written as a C interface. To implement the entire framework environment some tools were used: LLVM with Clang for language back-end and front-end, respectively; Boost C++ library and the Threadpool concept for the concurrent under the hood support; Bullet Physics and the Ronin Engine [Alves 2010] for the engine steps support; a modified Eclipse CDT tool to use as a programming environment.

Both C and C++ languages are used by lots of engines, because they provide easy access to hardware, GPU included. To access the graphics hardware there are some specific languages which interfaces are accessed with the used graphics layer. GLSL, HLSL and NVidia Cg are some examples. Other languages provide standalone interfaces, as NVidia CUDA and the OpenCL APIs. The C grammar is simple because the language was developed to be a portable assembly. That way, we utilize C grammar simplicity and explicit low-level parallelization with a robust C interface for C++ codes.

LLVM is a virtual machine and a framework for building compilers. The LLVM has its own assembly, similar to CISC architecture instructions, called Static Single Assignment (SSA) [Lattner and Adve 2004]. The Clang tool is a front-end for C-like languages, including Objective-C and Objective-C++. It is capable of code
generation for the LLVM, but only its C part is complete and somehow stable, even though LLVM/Clang self compiles, the C++ part of this front-end is not yet considered usable, i.e., the API is still in constant modification.

The Boost library has abstractions for multithread, filesystem, and metaprogramming. It is used on the underlying library that encapsulates the C interface for both the engine and the parallel structures. The Threadpool concept that uses the Boost multithread library is used on this library as well and supports multiple threads management. The works to run are called tasks and these are scheduled to different running threads.

The rendering part of the Ronin engine is complemented with the Bullet Physics solid bodies simulation [Bullet 2009; Boeing and Bräunl 2007]. Both Bullet and Boost are written in C++ and are common commercial solutions. These libraries work together under the same interface, illustrated in figure 1 as a UML component model. The Thread Mixer is a library which includes the Threadpool and control structures. The RN part provides a C interface for the Ronin engine and Bullet Physics.

1.1 Problems and solutions

During the framework development some problems were found. Here we discuss the major problems and their respective solutions.

C is an old language and there are lots of complete compilers for it. It was decided then, that the Clang front-end was to be extended for the upcoming grammar modifications implementation. Clang is both stable and robust for C and it was considered a good starting point.

The extended grammar predicted lots of statements that executed asynchronously, this brought the need of an efficient and easy way to execute statements in parallel. Therefore, this was solved without programmer efforts by detaching the statement from the function code and putting it in another function. Remarks about this solution will be made later on.

For the real-time part of the solution, it is used a POSIX-like exception system that is thrown when the deadline is not achieved in time. This solution was divided in two different instructions. One has both the periodicity and deadline values and the other only the deadline one. These two instructions provide the possibility to bring imprecise computation instructions that could be used for level of detail management.

Providing the entire concurrent tools as language instructions is not wise because the amount of work for the front-end is huge and the result is mostly the same as a library solution. But, if specific types of commonly code used in parallel algorithms are studied, then some interesting instructions can be implemented. That way, five different instructions were defined.

1.2 Techniques

This work covers some wide range of different techniques, from parallelization and real-time to physics treatment. These are described in this session, as a overview of the work different solutions.

The Bullet Physics library was used and it implements a fast GJK collision detection algorithm [van den Bergen 1999] and parallelizable broadphase and integration steps. Also, it has specific optimized code for parallel architectures as GPUs and Cell SPUs, performing broadphases on both multi-core and vectorized hardware. The integration with this library is done transparently with Ronin engine. This approach makes the physics interface of the framework independent from the underlying Bullet Physics implementation.

The Boost Threadpool concept uses the Boost libraries mutex and condition structures for implementing a low-level thread manager simple scheduler, that can be dynamically resizable (thread number) and support two simple scheduling policies: FIFO and LIFO. Even though this simplicity implies in some restrictions of tasks dependency, this makes the code faster and heavily parallelizable achieving high usage means for greatly parallelizable codes.

The Ronin engine uses an octree based clipping and culling algorithm with a BSP collision detection, promoting real-time complex structures support and optimized rendering for real-time graphics [Akenine-Möller et al. 2008; Jiménez et al. 2001]. The octree hierarchy is optimized using its loose version [Akenine-Möller et al. 2008].

This work is divided in the following sections: Section 2 shows the language keywords and their semantics; Section 3 talks about parallel and real-time code; Section 4 describes the Ronin engine; Section 5 exemplifies some solutions with its remarkable data; Section 6 shows our conclusions with this work.

2 Language definitions

To provide the needed support the five language instructions were defined for parallel processing and two to provide a soft real-time environment capable of imprecise computation. After each keyword definition we show the extensions we did to the ISO C99 [ISO 1999], in the BNF notation. For this BNF part the ID stands for IDENTIFIER and TPN for TYPE_NAME. Also, the Facade-lib was defined and its integration with the current CEx apps is illustrated in figure 1.

2.1 Concurrency keywords

The five keywords defined and their semantics are explained in this section.

2.1.1 parallel

It defines that the target statement will be executed in parallel. The parallel indicates that the internal scheduler will treat the parallel call the way it finds better. That means that the caller holds no control about the parallelized block execution. It is designed for usage in not emergencial or almost optional calls, which won’t change the final result drastically.

\[
\text{function_definition_async} :: = \text{PARALLEL}\text{\ function\ definition\ sync}
\]

2.1.2 async

Function and block modifier to execute the target in an specific job or thread. Indicates that the block may be used asynchronously, that is, there is no relation the executed code, its current condition and ending.

\[
\text{function_definition_async} :: = \\
\text{ASYNC} \\ ( \text{ID} ) \ 1 \ \text{function_definition_sync}
\]

2.1.3 atomic

Warrants atomic access to the variable\(^1\). Binary, logic and comparison operations will be atomic. The behavior for other operators is undefined.

\[
\text{type_qualifier} :: = \text{ATOMIC}
\]

2.1.4 parallel_for

Loop structure where the syntax is equal to a “for”. All iterations are possibly executed in parallel. There is no type of dependency explicit notation between the parallel executions. If necessary, the programmer can manually program such codes using the intermediary library. This instructions is based on OpenMP [Dagum and

\(\[\text{signed}\text{ and unsigned}\{\text{char, short, int, long}\}\] \)

---

\(\)
Menon 1998], although it resembles the *cilk_for* [Leiserson 2009; CIL 2009], the scheduling and synchronization styles are different.

```c
<iteration_statement> ::= 
    PARALLEL_FOR ("i" TPN ID ": i" ID ') <statement> |
    PARALLEL_FOR ("i" TPN ID ": i" CONSTANT ' , ' CONSTANT ') <statement> |
    PARALLEL_FOR ("expression_statement 
                  expression_statement 
                   ')' statement |
    PARALLEL_FOR ("expression_statement 
                  expression_statement expression 's') statement
```

### 2.1.5 synchronized

Supplies automatic synchronization of a statement, accessed by multiple threads. One structure must be passed as its argument and the semantic of the instruction depends if the variable is either a mutex, semaphore, condition or barrier.

```c
<synchronized_statement> ::= 
    SYNCHRONIZED ("i" identifier_list ') ' <statement>
<synchronized_statement> ::= <synchronized_statement>
```

### 2.2 Real-time keywords

The real-time instructions described in this section support soft real-time programming in our platform. Thus, it is necessary to integrate two different types of real-time blocks: the blocks that will be executed once and which must perform under a deadline time and a periodical one, that executes always under a deadline.

#### 2.2.1 do_rt

This keyword indicates that the target statement must be executed in a specified time (deadline). That way, the block must execute in the maximum time, as specified. The default behavior, if the real-time signal is untreated is to abort the program, as a POSIX *kill* signal. Such specification is similar to the POSIX [IEEE 2004] signals. Real-time exceptions must specify which are its handlers. This is done via the intermediary library.

```c
<rt_statement> ::= 
    DO_RT ("i" <constant_expression> ') ' <statement>
<statement> ::= <rt_statement>
```

#### 2.2.2 periodic_rt

It is semantically and syntactically identical to the *do_rt*. The difference is in the fact that the block is treated as a periodic one. It must be executed before the deadline, for each period. It is similar to attach a *while* with a *do_rt*, but in this last case the same code could be executed more than once in the same period.

```c
<periodic_rt_statement> ::= 
    PERIODIC_RT ("i" <constant_expression> ') ' ("i" <constant_expression> ') <statement>
<statement> ::= <periodic_rt_statement>
```

### 2.3 Facade-lib

Since we implemented the language with the Clang frontend, and most of our underlying libraries are written in C++. Then, it was necessary to provide these library services with a C interface. Also, the Ronin engine is written mostly in C++ and so a C interface for it was necessary. Both these interfaces together are what we call the Facade-lib. The concurrent part provides abstraction for mutex, semaphore, barrier, condition, task, job.

Task is some computation that must be done and can be done in a parallel way in the same job. Jobs are sets of tasks that contribute to a greater computation. In the case of a *parallel_for* each iteration creates a task under the same job. The *async* and *parallel* keywords creates tasks to be executed in the target job, passed as an identifier. When the programmer wants to execute these jobs in new threads, this library defines a *LONG_JOB* identifier, which means that the parallelized statement will run in a new thread. That is, it is equal to a thread spawn instruction.

### 3 Parallelism and real-time overview

Scheduling can be optimized for MIMD by the correct processor work distribution algorithm. In a work sharing algorithm, the processors share threads with the others when new threads are allocated, but on high usage the threads tend to migrate between processors increasing latency due to scheduling. In work stealing algorithms, the processors tries to steal the new threads and so in high usage cases the threads tend to stay in the same processor.

The Threadpool scheduler is compliant with the work stealing algorithm and allocate all threads for processors in a work stealing scheme [Blumofe and Leiserson 1999].

Amdahl’s Law [Amdahl 1967] states that an algorithm running on n cores with a number p for the total percentage of parallel parts of an algorithm, and s is its counterpart, then there is the following relation:

\[
\frac{1}{s + p} = \frac{1}{s} + \frac{1}{p}
\]  

For example, if s was 50% and then the n was considered infinite, then the maximum amount of speedup would be simply:

\[
\text{Speedup} = \frac{1}{s}
\]  

Which leads us to the result 2. That means that the maximum parallelization speedup of an algorithm is determined by its serial parts. In this case, the maximum speedup of a parallel version of this algorithm would be 200%.

The implemented instruction set for parallelism consist of five keywords which can be used with current C and C++ libraries. The *parallel_for* instruction creates a task under the same job. The *async* and *parallel* instructions spawn new tasks from the target statement and the synchronized shows a point where all workers will synchronize under a semaphore, condition, mutex or barrier. The last instruction is the atomic instruction which turns all invocations of the variable into atomical lock-free operations. In our LLVM implementation we prioritized the first four, excluding the atomic keyword instructions implementation. Some of these instructions are shown on the producer-consumer algorithm shown in figure 2.

If we have a 100% parallelizable code, where S is a startup time, including thread overhead creation time, T is the task creation time,
\( P_n \) is the number of processors, \( L_t \) is the worst case of loop execution unit time and \( N \) is the number of iterations. Then we have the following equation:

\[
    t = \frac{L_t \cdot N}{P_n} + N \cdot T_s + S
\]

This lead us to the worst execution time of the algorithm, \( t \). Notice, that the optimizable parts of the code, would be the thread creation time. This overhead can be diminished with code optimization, but in our case, we also execute parallelizable for iterations, while the for condition and task creation runs parallel with the already allocated tasks. Therefore, we achieve higher throughputs in concurrent loops because we hide the latency of task creation with the task execution. Our parallel loops executes far better when there \( L_t \) is larger. That is, we achieve higher efficiency on massively parallel and CPU hungry algorithms.

Real-time programming algorithms, on the other hand, involve either deadlines and periodicity or both. Most real-time systems are sensors observing systems that follow the idea of environment-sensor-control-actuator [Buttazzo 2004]. However, when game programming is taken into consideration, the main reason for providing real-time instructions would be for render flux control. That way, when wanted, the current rendering can be discarded or modified for achieving the rendering deadline, maintaining the rendering step. These, could be used as well for level of detail adjustment, adaptive graphics for the system and imprecise computation support. The POSIX like signaling of both do_rt and periodic_rt instructions makes easier to detect the deadline skip. Suppose that the main loop of a game is inside a periodic_rt instruction and that when the deadline is skipped, the current frame objects are rendered as boxes. This effect can be seen in figure 3.

Hard real-time scheduling can use only up 70% of the processing power [Liu and Layland 1973]. Then, at least 30% percent of all computing power won’t be used for useful computation in a hard real-time environment. Of course, this cost is prohibitive for gaming software and they don’t have to perform always under a specific deadline. An average rendering of frames per second of thirty or sixty are common, thus they are desirable, but not really a game purpose. Therefore, we only support soft real-time with our instructions, not applying some common techniques seen in hard real-time environments, like process priority inversion. Even our real-time exception system can’t be considered reliable in that way.

These instructions and the parallel library are complemented by the Ronin engine and they are implemented in the Thread manager library.

### 3.1 Thread manager

The Thread manager (TM) is part of the Facade-lib. It implements the functions that the compiler backend will call to implement the concurrency and real-time instructions. The Thread manager consists of a C interface that implements mutex, semaphore, condition, barrier, tasks and jobs. Every Job runs on at least one thread and contains tasks. A Task run on a thread, dynamically scheduled, of the Job it belongs. We needed a dynamical thread scheduler and the Threadpool BOOST concept was chosen. There are other commercial and similar applications like KDE Threadweaver [Boehm 2008] but they are either dependant on other libraries or overly complex. To maintain simplicity and attain high Task throughput we used Threadpool. That way, our Job concept has a built-in threadpool which schedules the Jobs tasks over time. A new task will only be scheduled when another one has finished its execution. That way, the Task abstraction is very different from a thread which can be directly mapped to a Job running only one task (LONG_JOB).

Mutex, semaphore, condition and barrier are all implemented with the the BOOST library condition and mutex abstractions. These are used directly with the manager interface, with calls implemented by the compiler backend.

### 3.2 Instructions and OpenMP

Our defined instruction provide syntax-level real-time instructions and parallel instructions. There are other works that do the same, like the OpenMP model. OpenMP is an API for multi-platform shared-memory parallel programming in C/C++ and Fortran, implemented in a lot of compilers and is independent of target platform.

OpenMP was developed aiming High Performance Computing. It has a great set of instructions and is, because of that, not a simple API. The CEx language was built aiming parallel processing as OpenMP, but with a different goal: we wanted the programmer to have a easy way of using parallelism instructions without having the obligation of exposing scheduling dynamics and internal blocks specifications. In this manner, OpenMP is a more complex and complete tool, but with it is needed a high-level of formality to write the code. The figure 4 depicts that formality with a simple example of vector add method. As a standard, OpenMP is tested and reliable but as a practical daily usage tool we believe that CEx can be a more programming time effective tool. Also, we wanted the interference with the current code syntax to be minimal, so scheduling and block size were hidden instead of exposed and this resulted in a major difference between both APIs.

It may be highlighted as well that it has no support, in any way, to real-time instructions, because OpenMP's concept model does not include real-time signals. Although, that may be possible, while using OpenMP in a real-time environment, e.g. RTOS, it is essentially different than using a concept model that is target independent and covers real-time instructions.

The next session describes the Ronin engine architecture.
#include <omp.h>

void vectorAdd(float *a, float *b, float *c) {
    #pragma omp parallel shared(a,b,c,chunk) private(i)
    {
        #pragma omp for schedule(dynamic,chunk) nowait
        for (i=0; i < N; i++)
            c[i] = a[i] + b[i];
    }
}

int main () {
    ...
    vectorAdd(a, b, c);
    ...
    return 0;
}

Figure 4: Vector addition example performed with OpenMP instructions in C++. OpenMP exposes scheduling policy and block size.

4 Ronin engine

In this section, we present the architecture of the 3D game engine Ronin. We will explain the physics, collision detection and rendering modules and the basic architecture of the engine.

4.1 Basic architecture

The engine’s basic modules are: Rendering, Physics and collision detection. For physics, we use the Bullet Physics engine, while the rendering and collision detection are implemented by the engine itself.

4.2 Bounding volume hierarchies

The engine uses octrees and Binary Space Partitioning (BSP) trees for rendering optimization and collision detection. The rendering part uses a quadtree or an octree to optimize view clipping and culling. The octree and quadtree implementations use the loose implementation, as in [Akenine-Möller et al. 2008]. The collision detection uses BSP trees to store complex objects geometry.

4.3 Rendering

The rendering module was implemented to be flexible, allowing rendering in single thread or multi thread.

4.3.1 Single thread Rendering

The sequential version of the rendering module goes through the scene tree (octree or quadtree), clipping the nodes and objects against the frustum. The objects that intersects the frustum are then rendered.

4.3.2 Multi thread Rendering

In the multi thread version of the rendering module, a pool of visible objects is populated by one or more clipping threads (worker threads that perform clipping on octree/quadtree nodes and objects). The rendering thread draws the objects in that pool without the need of clipping against the frustum (which was already done by one of the worker threads).

4.4 Collision detection

The collision detection module has a different propose from physics collision detection. Instead of detecting all collisions, for collision reaction, the collision detection module target the collisions of a single object. It also allow the use of an arbitrary geometric object to be used to detect collisions. The collision detection module is intended to be used direct by game logic.

This module uses a regular BVH to organize the objects in space. Each collidable object has a CollisionHandler, which hold the collision geometry (does not need to be the same of physics nor the rendering) and make the collision test. For complex objects, its geometry is stored in a Binary Space Partitioning tree.

4.5 Physics

The physics module uses the Bullet [Bullet 2009] physics engine. This module implements the facade pattern [Gamma et al. 1994], to ease the use of the physics engine and better integration with the rest of the engine.

4.6 Other modules

Among the other important modules there are the modules of Input and Shaders loading. The Input module supports basic events like mouse, keyboard, joystick and events related to window management (focus and resize). The Shader loading uses the abstract factory pattern [Gamma et al. 1994], allowing factory interfaces to many different shading languages.

5 Prototypes

Some prototypes were implemented to test the language concurrency support and graphics library. These were performed on the Setups observed in table 1. While the first setup has a dual core processor the second has a quadrupe core processor.

5.1 Parallelization examples

Concurrent programs examples are a simple way to show the parallelization capabilities of the concurrency library. Three benchmarks
were picked: a recursive Fibonacci’s algorithm, a parallel Mandelbrot set render.

The Mandelbrot example results are observed in table 2a and illustrated in figure. The parallel code in the dual core, runs almost two times faster as the serial one, but the quadruple core setup runs only 255% faster. This happens because the communication between the CPU and GPU starts to bottleneck the application parallelizable code. This is a practical observation of equation 2, which is Amdahl’s Law expression.

The same happens with the Fibonacci code, but this example suffers from task creation latency, that is, the parallel execution of each loop is not complex enough to show greater gains. For example, in the Setup 1 we observe that the transition from 3 to 6 threads yields almost no gain. While in the Setup 2 it supplies a significative gain. If we used more than 6 threads there wouldn’t be much gain, and more than that, the thread creation overhead could actually make the algorithm perform slower.

Both results are in table 2b and shown in figure 5. In this graphic we can see that when the thread number increases above 1.5 * processors the gain is not great. For example, in the Setup 1 we observe that the transition from 3 to 6 threads yields almost no gain. While in the Setup 2 it supplies a significative gain. If we used more than 6 threads there wouldn’t be much gain, and more than that, the thread creation overhead could actually make the algorithm perform slower.

### 5.2 Eclipse integration

To support the CEx language it was needed an Integrated Development Environment (IDE) with support to code refactoring, method generation; code completion, lexical, syntactic and semantic analysis, Syntax highlighting, semantic navigation, version control tools and multiple project management.

After evaluation of these parameters it was implemented as an extension of the CDT Eclipse plugin. That way, it was constructed an development editor with real-time effects and code test could be constructed this way. In fact, plans for an editor plugin are in the project roadmap and a fully functional editor is a long-term objective. Finally, the engine physics support needs to be improved to support deformable objects, already used in many games.

#### 5.3 Imprecise computation

Automatically adjusting the level of detail due to scene complexity, sluggish rendering and distant or fast objects is commonly used in game engines. As a proof of concept, we used the Mandelbrot set example to show how the real-time signaling instruction can change the current rendering frame. A half-rendered scene is illustrated in figure 7. That technique could be used for an automatic scene manager, so the game complexity would be adapted to the computing capacities of the computer. That is, the game could try to adapt the game video options according to the achievement of deadlines or not with our signaling instructions. The object to box swap example also highlight another usage that is related to imprecise computation.

### 6 Concluding remarks

Implementing a concurrent and real-time basis is a solution for building a platform for graphics programming. Although our implementation is not complete, the prototypes implemented with these show that such solution is possible. The parallel prototypes benchmarks exposes that the communication between CPU and GPU is a bottleneck for some applications as much as the non-parallelizable parts of algorithms. Optimization in the communication should be considered in these cases. Also, the results show that the processing results are pretty linear in scalability, therefore our scheduling efficiency is comproved.

Game development is a complex subject, but the current platform can supplement most parts of it, aside from sound support and artificial intelligence step interface. Both can be added in the game code, so that is not a problem. Other than that, dynamic code with LLVM can generate code and execute just-in-time. A complete game development editor with real-time effects and code test could be constructed this way. In fact, plans for an editor plugin are in the project roadmap and a fully functional editor is a long-term objective. Finally, the engine physics support needs to be improved to support deformable objects, already used in many games.
The language extension was implemented almost on its full and its missing parts are more syntax simplifications than necessities. Then, aside from improvement of the current implementation with Clang, it is mandatory that an extension similar to the purposed one is made for C++. Also, a debugging tool for deadlock and concurrency bottlenecks detection is needed because as the complexity of the problems and programs grows, so does the interest in tools like these. It can be noted that some third-party tools for profiling and debugging as the Valgrind framework may already be used.

The language also needs improvements for shader programming. Instead of loading a shader as it is currently done, the language should have its own shader syntax, integrated with the rest of the language, making easier to game developers to make use of such technology and granting portability.

References


Figure 8: Physic simulation of a thousand boxes. In I all boxes drop from the sky. At II they are stable. In III they suffer a huge force to them center and in IV and V they repel each other and fall into rest, respectively.

(a) Mandelbrot set complete render. The almost in set values are highlighted with darker tones.

(b) Test game implemented in CExs. The AI is computed in parallel using the language instructions. The enemies throw balls into the player direction with some swarm intelligence.

Figure 9: Rendering tests with CEx and Facade-lib