Hellaynnea Consulting
hellaynnea@gmail.com

NetMultiPlayer

Multi-Threaded Video Rendering

(Ref.: NMP_MTVIDEO_ YL)

April 10th, 2006

Table of Content

Introduction

The purpose of this document is to present the state of my work in terms of graphical engine adapted for multi video channels rendering. I have been faced the last three years to different problems and constraints that led to the choices I had taken today. The literature in this field is very spare as most of the articles I could find where related to single video rendering (DivX player, DVD player, etc.). No article was focusing on how to render multiple video channels on an optimized manner, introducing multi-threaded programming at the same time.

In this document, I will first give a guide to safe multi-threaded programming. Then I will describe the architecture of a graphical engine supporting multi video channels rendering. I don't want this document to be exclusive on the technology to be used, however, I have been working with a lot of different technologies and the one presented here seemed the most suited for this kind of work.

References

- The software development plan (SDP) of the Net Multiplayer software system. This document gives a detailed schedule of the project workload.

- The software architecture document (SAD) of the NetMultiPlayer software system. This document describes the full architecture of the project and interactions between the different pieces of software involved.

Multi-threaded Programming Guide

Actors Definition

(Ref.: www.wikipedia.com)

Thread

A **thread** is short for a *thread of execution*. Threads are a way for a program to split itself into two or more simultaneously running tasks. Multiple threads can be **executed in parallel** on many computer systems. This *multithreading* generally occurs by **time slicing** (where a single processor switches between different threads) or by **multiprocessing** (where threads are executed on separate processors). Threads are similar to **processes**, but differ in the way that they share resources.

Mutual Exclusion (Mutex)

Mutex algorithms are used in concurrent programming to avoid the concurrent use of un-shareable resources by pieces of computer code called **critical sections**. Basically, a mutex is used as a lock to prevent other threads to access the protected portion of code (a critical section of the code). In OOP, a mutex is usually embedded in a class providing two basic functions: lock and unlock. The lock function allows a thread to ask for an access to the critical section. This access is granted only if the lock is available, otherwise, the thread is paused until it freed. The unlock function is called by a thread to indicate that it is no more in the critical section.

Synchronize Class

The synchronize class is a super class of Mutex I had to create. The Mutex class provides only an exclusive access to the resources regarding to the way a thread may want to access to them. This means that only a single thread can access the resources at a time. But, in reality, threads have two ways of accessing a resource: a **read** access (no modification) and a **write** access (with modification). To model this behavior I have created the **Synchronize** class. This class is using a simple algorithm:

```
/// synchronized class
class Synchronized
{
      Mutex data_mutex_;      /// to protect the class' data
      Mutex mutex ;           /// to protect the critical section
      long  lock_count_;      /// read access count
      bool  X_lock_;          /// exclusive lock flag

      public :

            /// class constructor
            Synchronized()
                  :lock_count_(0)
                  ,X_lock_(false)
                  ,ref count (0)
            {
            }

            /// virtual destructor
            virtual ~CSynchronized()
            {
            }

            /// get lock count
            long lock_count(void)const
            {
                  return lock_count_;
            }

      protected :

            /// lock class access
            inline void __lock(bool bXLock)const
            {
                  if( bXLock )
                        x lock();
                  else
                  {
                        mutex_.lock();
                        data_mutex_.lock();
                        ++lock count ;
                        data_mutex_.unlock();
                        mutex_.unlock();
                  }
            }
```

```
/// unlock class access
            inline void __unlock(bool bXLock)const
            {
                  if( bXLock )
                        __x_unlock();
                  else
                  {
                        data_mutex_.lock();
                  if( lock_count_ > 0 )
                              --lock_count_;
                        data mutex .unlock();
                  }
            }

      private :
            /// Xclusive lock
            inline void __x_lock()const
            {
                  mutex_.lock();
                  while( lock_count_ > 0 )
                        _sleep_(1);

                  data_mutex_.lock();
                  X lock  = true;
                  data_mutex_.unlock();
            }

            /// Xclusive unlock
            inline void   x unlock()const
            {
                  data_mutex_.lock();
                  bool unlock mutex = X lock ;
                  if( X_lock_ )
                        X_lock_ = false;
                  data_mutex_.unlock();

                  if( unlock mutex )
                        mutex_.unlock();
            }
};
```

Read Access

The read access is done by locking the critical section mutex, incrementing a counter and unlocking the mutex right after. This means that the mutex will not be held during the entire access to the critical section but only at the beginning to signify that the thread is requesting a read access. Once the thread is done with the resources, it calls the unlocking function which decrements the counter. With this simple mechanism, several threads can access to the same resources at the same time as soon as they only intend to read the resource's content.

Write Access

The write access is done by locking the mutex and waiting for the read access threads to stop their activity (counter is null). The fact that we wait before returning makes sure that the thread won't be able to perform any writing activity unless all the reading threads are done with the resource. Moreover, keeping the mutex locked prevent any other threads to request a read or write access for the same resources. When the writing thread is done it calls the unlock function which unlocks the resource's mutex.

A programmer can derive any class from Synchronize to have access to these mechanisms easily.

Remark:

Note that the counter is protected by a data mutex. The data mutex is intended to protect the content of the class in a multi-threaded environment. Indeed, on a multi-core system, two cores can, at the exact same time, decrease the counter (unlock function); so that instead of decreasing the value per two, it is only decreased per one. This will result in an invalid state of the counter variable, locking the next write-access request in a sleep state as the counter will never be decremented to zero. And, all the next lock requests will join this sleep state, the program will ultimately freeze.

Auto Mutex Class

The auto-mutex class has been developed for the purpose of this program. This class is a super class of Synchronize. It exploits the scope of a variable to lock and unlock a mutex easily. I have created a class with two constructors: one handling a mutex and one handling a Synchronize class. The algorithm of the auto-mutex class is very simple. The creation of an auto-mutex variable requires a link to an actual mutex or Synchronize class. When the constructor is called, it locks the mutex or the synchronized class. When the destructor of the variable is called, it unlocks the mutex or the synchronized class.

```
/// a mutex selft-managed
class AutoMutex
{
	Mutex * mutex_; // our mutex pointer
	const Synchronized * synchronized_; // synchronized pointer
	bool Xlock ;

	public :
		// lock access when initialized
		AutoMutex(Mutex * m)
			:mutex (m),synchronized (0),Xlock (false)
			{ if( mutex_ )mutex_->lock(); }

		// lock access when initialized using synchro class
		AutoMutex(const Synchronized * s, bool Xlock)
			:mutex (0), synchronized (s), Xlock (Xlock)
			{ synchronized_->__lock(Xlock_); }

		// unlock access when out of scope
		virtual ~AutoMutex()
			{
				if( synchronized_ )
				{ synchronized_->__unlock(Xlock_); }
				else
				{ if( mutex_ ) mutex_->unlock(); }

			}
};
```

Using such a class simplifies a lot of things. Indeed, a programmer just has to create an **AutoMutex** variable at the top of a function using a shared-resource. The constructor will then be called locking the mutex. Whenever the function will return, the destructor will be called unlocking the mutex automatically. No need for the programmer to check that all the return functions have an associated call to the *mutex.unlock()* function.

```
Mutex my_mutex;
Synchronize my synchro;

void foo1()
{
	AutoMutex auto_mutex(&my_mutex);

	[ACCESS SHARED RESOUREC]
	if ( condition1 )
		return;
	[ACCESS SHARED RESOUREC]
}
```

```
void foo2()
{
      AutoMutex auto_mutex(&my_synchro, true);

      [ACCESS SHARED RESOUREC]
      if ( condition1 )
            return;
      [ACCESS SHARED RESOUREC]
}
```

Deadlock

A **deadlock** refers to a specific condition when two or more processes are each waiting for another to release a resource, or more than two processes are waiting for resources in a circular chain.

My Multi-threaded Programming History

Four years of multi-threaded programming experience really changed my vision of programming in general. More specifically, it changed the way I conceived my architecture. At the beginning, I was thinking that a super class, called *Brain*, managing and synchronizing everything was a good solution. I slowly moved towards a more modular system where all the parallel algorithms would be performed by a different module (or core) of the system. This architecture forced each thread to be wrapped in a specific core and a core could not execute more than one thread (with some exceptions). Each core would then protect its own data, and access the other core's data by some exchange/communication protocol as if the other core was on another machine (which can be the case with distributed algorithms).

This refactoring of my thinking process forced me to come up with new programming rules for the core themselves. It became clear that it would not be possible to manage the mutex and their associated resources properly if threads could access other threads resources directly taking care of the locking mechanism externally of the resource's owner.

Example:
We have two threads T1 and T2 sharing a resource R. R does not really belong to any of the two threads but both threads need to use it. R is given a mutex M to protect its access.

A typical algorithm I was coding years ago was:

```
<T1 or T2>
[…]
M.lock()
R.readSomething()
R.writeSomething()
M.unlock()
[…]
</T1 or T2>
```

This seems fairly simple and it is. But more problems arise when the threads have more than one mutex to manage. Indeed, deadlocks can then occur and both threads would be blocked forever. Instead of having to adapt the code regarding to the number of mutex there is, it was better to come up with general rules, more OOP-compliant.

Multi-threaded Programming Rules

This section intends to give general rules to guarantee a 100% safe multi-threaded program.

Rule 1: One thread = 1 Synchronize Class

One thread must be handled in one class (or core) and a class can only handle one thread. This class must be derived from the Synchronize class.

Rule 2: One Resource = 1 Mutex only

Instead of having to manage different mutex per resource, a resource should only be associated with a single mutex. The resource should be wrapped in a class whose public functions (to interact with the resource) are protected (see Rule 3).

Rule 3: Public functions must be protected

Given that public functions are intended to be used by external components, other threads included, all the public functions must use the class mutex.

Rule 4: Public functions cannot call other public functions

Given that public functions are protected, a public function cannot call another public function of the same class otherwise it would stay locked! Only one exception is accepted, is when the public function is not using the class resource but only used for sorting/testing purpose (switch and if). In that case, the function is only calling public functions regarding to the parameters passed to it. This means that this particular function must not be protected. It is possible to avoid such behavior by coding the sorting sequence in private functions.

Rule 5.1: Private functions cannot call public functions

Private functions are assumed to be called under protection by public functions or other private functions themselves called by protected public functions. This means that inside a private function, the critical section is protected. This also means that a private function cannot call a public function.

Rule 5.2: Private functions cannot use the class mutex

According to rule 5.1, the private function is already protected so it must not use the class mutex.

Rule 6.1: Exclusive lock – Write Access : No *const* keyword

All threads requesting a write operation on a given resource must use the exclusive lock of the synchronize class. Thus, the functions performing such operations cannot be declared as *const*.

Rule 6.2: Simple lock – Read Access : Use of *const* keyword

All threads requesting a read operation on a given resource must use the simple lock of the synchronize class to allow other threads to read the resource as well. Thus, the functions performing such operations must be declared as *const*. If the *const* keyword cannot be added, this means that the function or sub-functions are at some point performing a write operation. The code must then be refactored.

If a programmer follows these rules, the resulting code should be guaranteed thread-safe.

Video Channels Design

Presentation & Algorithm

It is not needed to describe in details the way the video channel works. Indeed, one just needs to know that it is a loop fetching pictures and asking the graphical engine to display it. The important point, however, is that 25 of these channels are working in parallel, and that each one is using an MPEG4 decompression algorithm which consume processing power. This power consumption is inevitable so one need to optimize everything possible between the call to display and the picture actually displayed on screen.

```
procedure channel_thread:
      init()
      while brain.stop = 0 do
            if connected = 0 then
                  reconnect() // return only when a connection is made
                  connected := 1
            end if
            error = 0
            while picture_in_buffer() = 0 and error = 0 do
                  wait()
            end do
            if error = 1 then
                  connected := 0
            else
                  lock_picture()
                        picture := read_picture()
                  unlock_picture()
                  brain.display = 1
            end if
      end do
      cleanUp()
end proc
```

Optimizations

The purpose of optimizing the channel thread is to try to leave as much power as possible for the graphical engine. Indeed, displaying frames requires a lot of processing power, especially when the pictures must be stretched. The optimizations I will present below helped in achieving this purpose so that the channels processing needs are stuck to the processing power needed to decompress the MPEG4 frames.

Memory Pool

The channel has a specific feature which is to keep track of the last 10 seconds of video in memory so that the user can go backward and pause the stream at any time. The frames are stored in a FIFO buffer. When the stream is in pause, the new frames are dropped and the buffer is frozen so that the user can walk through it. There are several ways to program such features nevertheless I decided to implement a memory pool. A common way would be to create a new buffer every time a new frame is fetched from the network and queue it in the FIFO. This method is not recommended as each frame is around a 1Mo large and allocating (and freeing) megabytes of memory 25 times per second multiplied by the number of active channels is not really feasible. Concerning the memory issue, such feature can only be activated when only a few channels are running on screen (less than four).

To overcome the memory allocation time overhead, I have implemented a memory pool. A memory pool is a large chunk of memory (several megabytes), allocated at the creation of the channel, once and for all (until destruction), and distributing pointers to specific slots in the pool. Basically it is a sort of array with variable cells' addresses. I added another feature to it which is that it is handling the FIFO directly in it by moving the pointers around instead of moving megabytes of data (*memcopy*). So the memory management and frame management are done at the same time.

This optimization decreased the amount of resources needed by a channel to process its data leaving more power to the graphical engine to perform channel's display.

```
/// memory pool class
class MemoryPool
{
      unsigned char * pool_;              /// pool pointer
      unsigned int      block_size_;      /// data block size
      unsigned int      block count ;     /// number of blocks

      public :

      /// pool constructor
      MemoryPool(unsigned int pblock count=1, unsigned int pblock size=1)
            :pool_(0),block_size_(pblock_size),block_count_(pblock_count)
      {
            if( block_size_ > 0 && block_count_ > 0 )
                  pool_ = new unsigned char[block_size_*block_count_];
      }

      /// pool destructor
      ~MemoryPool()
      {
            if( pool_ )
                  delete [] pool_;
      }

      /// block size getter
      unsigned int block_size(void)const
      {
            return block_size_;
      }

      /// block count getter
      unsigned int block count(void)const
      {
            return block_count_;
      }

      /// get specific block
      unsigned char * getBlock(unsigned int block_position)
      {
            return (block_position >= block_count_ ? 0 :
                        (pool_+block_position*block_size_));
      }

      /// get specific block
      unsigned char * getBlock(unsigned int block_position)const
      {
            return (block_position >= block_count_ ? 0 :
                  (pool_+block_position*block_size_));
      }
};
```

Display Flags

There are different ways of designing the system's core to display a channel's picture. The first way I was taking was that, every time the channel was ready to display, it was creating a copy of its picture buffer and sent it to the core. This design was not really resources consuming so I had to review it (see *Memory Pool*). Instead, when a channel is ready to display a picture it simply sends a message to the core. The core, when ready to display, the channel will requires an access to the channel's current picture buffer (mutex) and display it.

The message sent is simply the ID of the channel and the type of picture the channel want to be displayed (connection, error or normal picture). The core collects the messages in a queue that it cleans every time all the requesting channels have been displayed.

Such method shaved the copying time and the core was sure that it was always obtaining the last picture in the channel. Another advantage is that the core can "force" a channel's display at anytime, requesting the current picture buffer. This method is also interesting when considering the backward/pause/forward features. Indeed, the channel's just has to provide a different buffer (changing the pointer address) to the core. By doing so, I am suppressing most of the buffers in the display chain.

Graphical Engine

Presentation & Architecture

OpenGLUT

OpenGL itself is just the API to interact with the graphic card. To add more features to it, the OpenGL programmers have created GLUT. GLUT provides an API for window programming as well as OpenGL shortcuts. Nevertheless, the features of GLUT were not sufficient for me. Indeed, they are game-oriented. I have found a super-library called OpenGLUT which adds more features to GLUT and OpenGL. *(N.B.: there also exists an OpenGLEAN library superseding the OpenGLUT capabilities. But OpenGLUT was sufficient for me).*

Chain of Command

The widgets are organized in both horizontal and vertical architecture. This means that they have parents and children but are also gathered in containers. When a user event occurs (mouse motion, click, key pressed, etc.), the event is passed to the first widget of the list which either process it or pass it to its children. In the case of horizontal architecture, the event is passed to the widgets in its area (coordinates of the mouse). If no coordinates are specified, the event is passed to all the widgets stored horizontally.

Message Pump

The graphical engine functions in a thread. As such, it must follow the rules of the Multi-threaded programming guide described above. Furthermore, we have to consider that OpenGL requires specific programming design to function properly in multi-threaded environment. Each command issued to OpenGL is stored in the graphical card. Once we are ready to perform the rendering, we can “flush” the command queue to have our commands executed. OpenGL works with rendering contexts which are not thread safe. Each thread must render in different contexts or the programmer must design a specific architecture to allow multiple threads to interact with another thread’s context. I chose to create a message pump, exactly like the GUI. In fact, the OpenGL thread *owns* the rendering context and all other threads (and associated cores) can only queue

messages which will be processed by the main thread's message pump. We have commands for screen clearing, channel display, channel creation/destruction, etc.

GTK Interaction

I had to make OpenGL and GTK interact so that I don't have to program a whole GUI system in OpenGL. To do so, I went into the code of GTK to find out how the message pump was working so that I could simulate it in the OpenGL thread. Basically, at each loop (30 times per second), OpenGL is calling, before rendering, the GTK message pump for a limited time to allow the windows to be displayed and refreshed. The limited time is mandatory as while a user is pressing a mouse button, the button-pressed event is sent to the message pump, giving no rest for the other threads. Limiting the time allows the pump to quit so that we can render pictures.

Singletons

The singletons are classes that must be created only once during the entire life of the program. All theses classes are gathered in a Singletons class which returns a reference to it through getters.

Facade

The façade is the external interface of a module for the other modules. It is a pure abstract class. The façade must contain all the useful functions that external modules can need.

Controller

The controller is derived from the façade class. It is the core of the module. Each controller should be associated with a thread (cf. Multi-Threaded Programming Rules).

Widget Architecture

In this section we will describe the architecture of widgets I have developed to perform the graphical rendering of the video channel's pictures.

Widget

The widget is the basic element of the graphical engine. It is an abstract class that other graphical elements must derive from to be considered in the rendering process.

Each widget is identified with an id provided in the constructor, as well as graphical utilities like size, coordinates, color. The widget also has a link with its parent widget (if any). This allows the programmer to create hierarchies of widgets and can be used for a chain of command, passing commands that you don't process to your parents. The widget provides a function *isVisible* to determine whether it should be displayed or not. This *isVisible* function only concern the current widget not the parents or children. The size parameter allows the graphical engine to resize the widget when needed.

```
typedef std::list<unsigned int> widgetIdList_t;

/// widget selection type
enum eSelectionType
{
	eselection left,
	eselection_right,
};

/// graphical widget
class Widget{

	protected:
		/// viewport
		static const Rectangle * __viewport__;

		unsigned int id_;
		Size size_;
		Coordinates coords ;
		Color color_;
		Widget * parent_;

	public:

		/// link with the viewport
		static void _link_(const Rectangle * viewport);

		/// ctor
		Widget(unsigned int id=1,
				const Coordinates& coords=Coordinates(),
				const Size& size=Size(),
				const Color& color=Color());

		/// dtor
		virtual ~Widget();

		/// id getter
		unsigned int getId(void)const{ return id_; }
```

```
        /// get complete id (including parents')
        void getFullId(widgetIdList_t& id)const;

        /// size setter/getter
        virtual void setSize(const Size& size){ size  = size; }
        const Size& getSize(void)const{ return size_; }

        /// coords setter/getter
        virtual void setCoordinates(const Coordinates& coords)
              { coords  = coords; }

        const Coordinates& getCoordinates(void)const{ return coords_; }

        /// get full coordinates
        void getFullCoordinates(Coordinates& coords)const;

        /// color setter/getter
        void setColor(const Color& color){ color_ = color; }
        const Color& getColor(void)const{ return color_; }

        /// set widget's parent
        void setParent(Widget* parent){ parent_ = parent; }

        /// draw
        virtual void draw(bool selectionRendering);

        /// is widget visible
        virtual bool isVisible(void)const;

        /// this widget has been selected
        virtual void selected(widgetIdList t& selection,
                              eSelectionType selection_type);

        /// get widget selection rectangle
        virtual Rectangle getRectangle(void)const = 0;

    protected:

        /// draw
        virtual void __draw(bool selectionRendering)=0;

        /// update widget
        virtual void __update(void);

        /// this widget has been selected
        virtual void __selected(eSelectionType selection_type);

        /// begin drawing
        virtual void   beginDraw(bool selectionRendering);

        /// end drawing
        virtual void __endDraw(bool selectionRendering);

        /// translate if needed
        virtual void __translate(void);
```

```
            /// draw ourself if needed
            virtual void __selfDraw(bool selectionRendering);

      private:
            /// forbidden operator
            Widget & operator=(const Widget& widget);

};
```

Widget Container

A widget container, as the name suggests it, contains other widgets. It allows multiple widgets to be grouped together in a horizontal architecture.

```
typedef std::list<Widget*> widgetList_t;

class WidgetContainer: public Widget{

      protected:

            widgetList_t widgets_;

      public:

            /// ctor
            WidgetContainer(unsigned int id=1,
                              const Coordinates& coords=Coordinates(),
                              const Size& size=Size());

            /// dtor
            virtual ~WidgetContainer();

            /// draw
            virtual void draw(bool selectionRendering);

            /// get selection rectangle
            virtual Rectangle getRectangle(void)const;

            /// is widget visible
            virtual bool isVisible(void)const;

            /// this widget has been selected
            virtual void selected(widgetIdList_t& selection,
                                    eSelectionType selection type);

            /// add a child widget
            virtual void addWidget(Widget * widget);
      protected:

            /// draw
            virtual void __draw(bool selectionRendering);

            /// draw contained widgets
            virtual void __drawWidgets(bool selectionRendering);
```

```
	private:
		/// forbidden operator
		WidgetContainer & operator=(const WidgetContainer& container);

};
```

Extended Widget Container

The extended widget container is an array of widgets adapting the size of the widgets to fit the array cell's size.

```
/// overlay of widgets
class WidgetContainerEx: public WidgetContainer
{
	protected:
		Rectangle		rect_;
		unsigned int		cell_countX_;
		unsigned int		cell_countY_;
		Size			cell_size_;
		float			marginX_, marginY_;
		Rectangle		widget_rect_;
		float			content_deviation_;
		bool			translating ;

	public:
		/// ctor
		WidgetContainerEx(unsigned int id=1,
					const Rectangle& rect=Rectangle(),
					unsigned int displayX = 1,
					unsigned int displayY = 1,
					float marginX = 0.0f,
					float marginY = 0.0f,
					const Color& color=Color());
		/// dtor
		virtual ~WidgetContainerEx();

		/// get selection rectangle
		virtual Rectangle getRectangle(void)const;

		/// add a child widget
		virtual void addWidget(Widget * widget);

		/// draw
		virtual void draw(bool selectionRendering);

	protected:

		/// draw
		virtual void __draw(bool selectionRendering);

		/// draw contained widgets
		virtual void __drawWidgets(bool selectionRendering);
```

```
/// update
            virtual void __update(void);

            /// this widget has been selected
            virtual void __selected(eSelectionType selection_type);

      private:
            /// forbidden operator
            WidgetContainerEx & operator=(const WidgetContainerEx& container);
};
```

Texture

Data structure to represent the textures.

```
/// Texture class
struct Texture
{
      int width, height;
      unsigned char * rgb_data;
      std::string label;

      Texture(int w=0, int h=0,
                  unsigned char * data=0,
                  const std::string& _label="")
            :width(w),height(h),rgb_data(data),label(_label){}

      ~Texture()
            { if( rgb_data ) delete [] rgb_data; }

      bool isValid(void)const
            { return width>0 && height>0 && rgb data!=0; }

      private:
            //-*** forbidden operators ***-//
            Texture& operator=(const Texture&);
            Texture(const Texture&);
};
```

Texture Register

The texture register is a class collecting all the textures used by the program. It is in charge of loading them from files and managing their ids. For a complete description of the Texture Register see *Optimization.Texture Register*.

```
typedef unsigned int textureId_t;

/// texture in register
class RegisteredTexture
{
      private:
            unsigned int      users count ;
            textureId_t       texture_id_;
```

```
      public:

            /// generate a texture
            static RegisteredTexture * generate(void);

            /// dtor
            ~RegisteredTexture();

            /// texture id getter
            textureId t getTextureId(void)const
                  { return texture_id_; }

            /// users count getter
            unsigned int getUsersCount(void)const
                  { return users count ; }

            /// register a new user for this texture
            void registerUser(void)
                  { users_count_++; }

            /// unregister a user for this texture [true=no more users]
            void unregisterUser(void)
                  { if( users count  ) users count --; }

      private:
            //-*** forbidden operators ***-//
            RegisteredTexture();
            RegisteredTexture& operator=(const RegisteredTexture&);
            RegisteredTexture(const RegisteredTexture&);
};

/// Textures handler
class TextureRegister{

      common::utils::ControllerDB<std::string,RegisteredTexture> textures_;

      public:

            /// ctor
            TextureRegister();

            /// dtor
            virtual ~TextureRegister();

            /// is texture registered
            bool isRegistered(const std::string& name)const;

            /// unregister a texture
            void unregisterTexture(const std::string& name);

            /// register a new texture
            bool registerTexture(const std::string& name,
                                 const Size& pic_size);

            /// update texture data
            void updateTexture(const std::string& name,
```

```
                                    const Texture& texture);

            /// get texture id
            textureId_t getTexture(const std::string& name);

            /// convert picture size to texture size
            static Size _toTextureSize_(const Size& pic_size);

      private:

            /// generate a new texture
            static textureId_t __generateTexture_(void);

            /// release a texture
            static void __releaseTexture_(textureId_t id);

            /// get texture max size
            static int __getMaxTextureSize_(void);

            /// get Texture Size
            static int __getTextureSize_(int size);
};
```

Textured Widget

The textured widget represents the widget able to display a texture. It is linked to a texture register to be able to manipulate ids instead of strings or data pointers.

```
typedef unsigned int displayListId_t;

/// Widget handling textures
class WidgetTexture: public Widget
{
      bool init_;

      protected:
            /// texture register
            static TextureRegister * __register__;

            std::string name ;

            Size picture_size_;
            Size channel_size_;

            displayListId t   display list id ;

      public:

            /// link with a texture register
            static void  link (TextureRegister* reg);
```

```
            /// ctor
            WidgetTexture(const std::string& name,
                                    bool is_file,
                                    const Size& channel_size,
                                    unsigned int id=1,
                                    const Coordinates& coords=Coordinates(),
                                    float alpha = 1.0f);


            /// ctor 2
            WidgetTexture(const std::string& name,
                                    const Size& picture_size,
                                    const Size& channel_size,
                                    unsigned int id=1,
                                    const Coordinates& coords=Coordinates(),
                                    float alpha = 1.0f);

            /// overload size setter
            virtual void setSize(const Size& size);

            /// dtor
            virtual ~WidgetTexture();

            /// get widget selection rectangle
            virtual Rectangle getRectangle(void)const;

            /// update texture data
            virtual void updateTexture(const Texture& texture);

      protected:

            /// draw
            virtual void __draw(bool selectionRendering);

            /// this widget has been selected
            virtual void __selected(eSelectionType selection_type);

      private:

            /// init texture and display lists
            void __init(void);

            /// (re)init the display list
            void __initDisplayList(void);

            /// (re)create the display list
            void __createDisplayList(void);
};
```

Optimizations

Texture Register

To save space and increase rendering speed, I have created a texture register. The texture register manages all the textures used by the program and allows the widgets to share the same textures. The register is in charge of allocating/freeing the memory and loading pictures if needed. The widgets just need to use a texture id instead of having to manage the texture data directly. Most of the textures are also charged in the graphic card memory.

Texture Mapping and Sub-Textural rendering

In the video-game industry, the textures are loaded once and for all in the graphic card. In the case of our program, the textures are refreshed 25 times per second. This means that, 25 times per second, we need to transfer the new texture data to the graphic card. There are different ways of doing so but the OpenGL programming guide suggest that we use a function replacing the picture by parts instead of replacing the whole picture at once. It seems to have decreased the data transfer overhead.

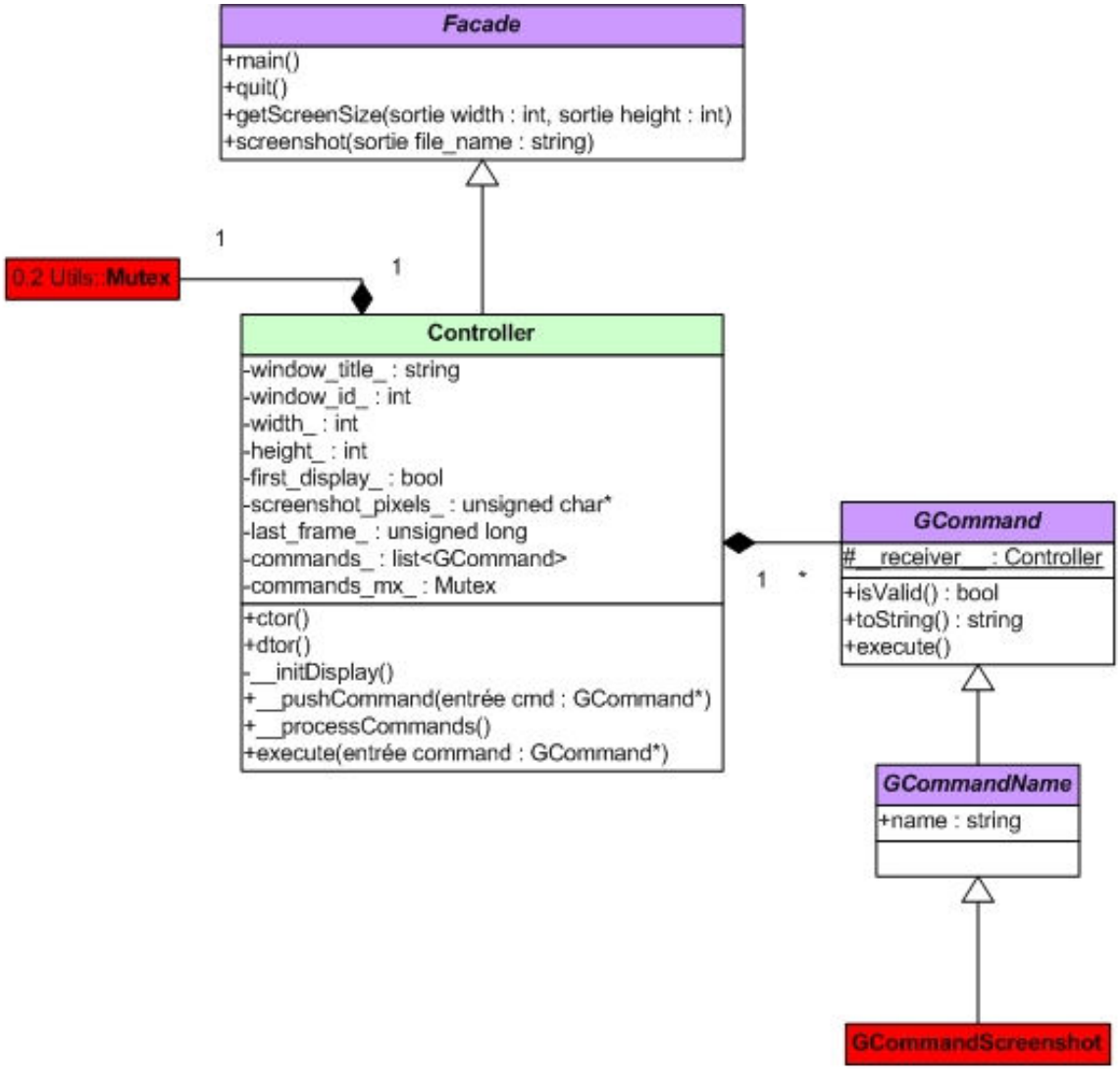
Display Lists

OpenGL has a feature called Display Lists which allows the pre-compilation of graphical instructions. These instructions are optimized, compiled and placed in the graphical card. Instead of having to call these instructions every time with the same parameters, one just calls the list. The fact that these lists are compiled, optimized and stored in the graphical card significantly increases the speed of execution. In this program, I have used display lists for texture rendering purpose. The textures addresses and size do not change once the channel has been created.

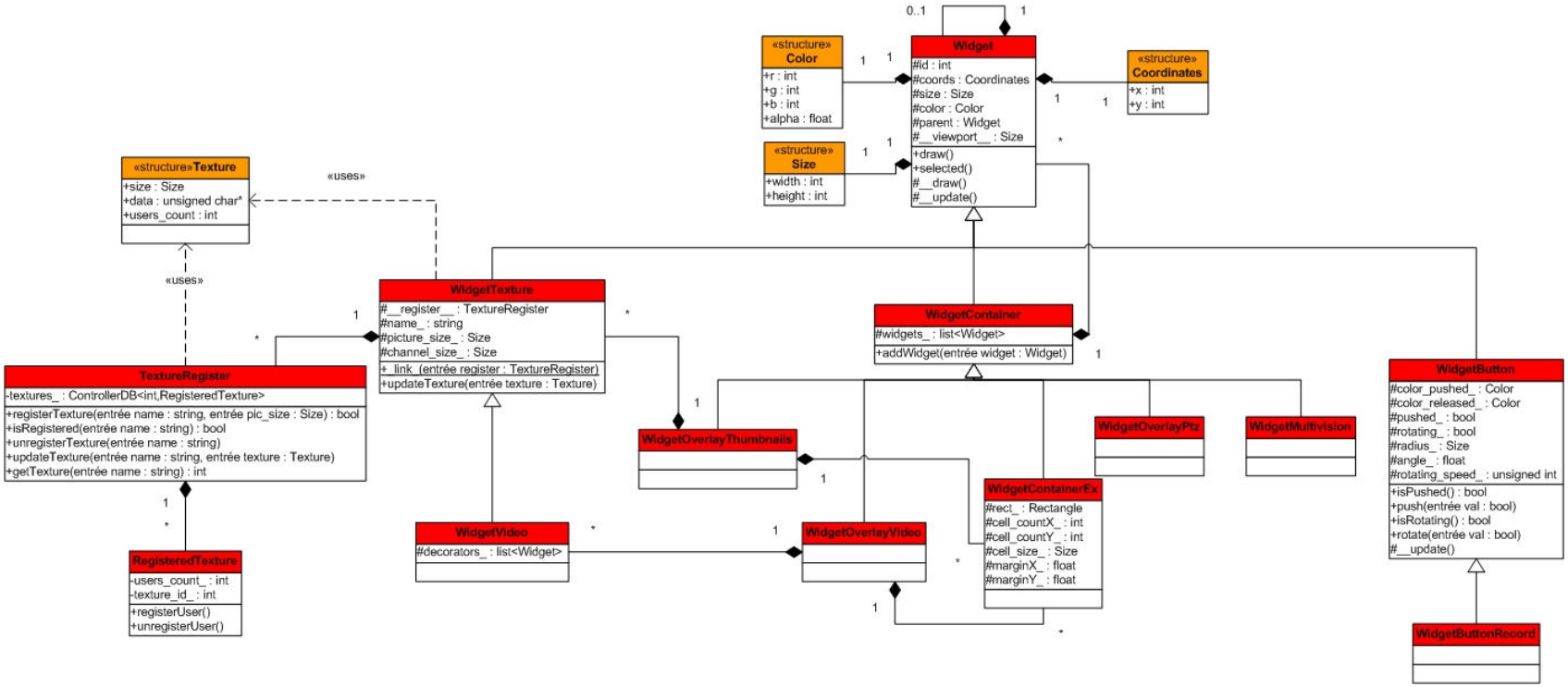
On Demand Display

The channels are only displayed whenever they as for it. The graphical engine thread doesn't redraw all the channels unless it is absolutely necessary.

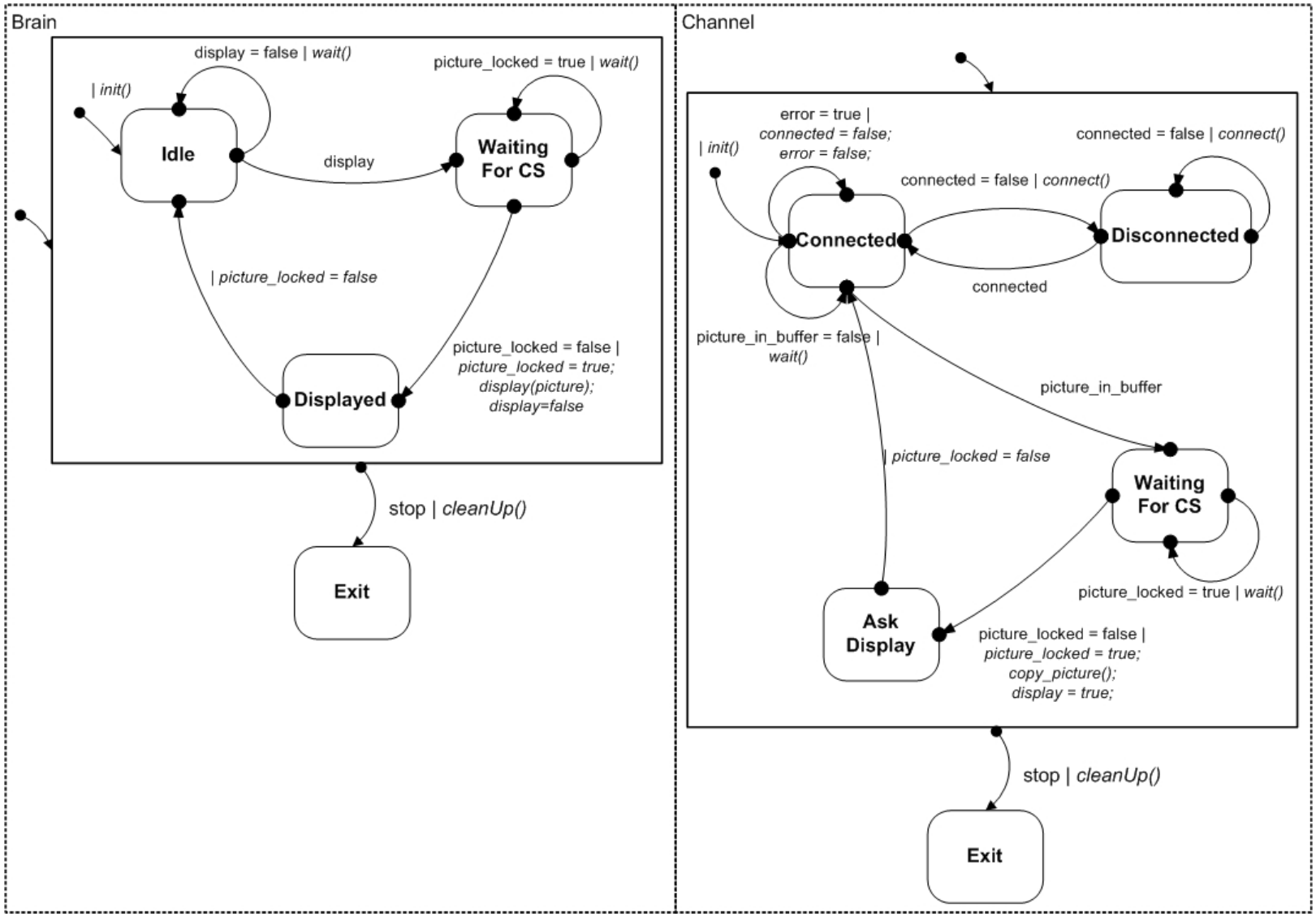
General Architecture



Widget Architecture



Channels & Graphics Interaction



Algorithm & Future Plans

```
/// graphic thread
procedure GRAPHICS::graphic_thread:
      init()
      while CORE.stop = 0 do
            /// process the commands list if needed
            if commands list != empty then
                  processCommands(commands_list);
            end if

            /// GTK message pump
            GUI.process gtk events();

            if CORE.isRenderingEnabled() then
                  CORE.render_scene(); // allow the core to render the scene
                  flush_gl_commands(); // execute all GL commands
            end if

            sleep();
      end do
      cleanUp()
end proc

/// GKT message pump
procedure GUI::process_gtk_events:
      loop_time = GetCurrentTime() // time reference
      while    gtk_events_pending()
            && GetCurrentTime() - loop time < 20  do //don't loop forever
            gtk_process_events();
      end do
end proc


/// scene rendering
procedure CORE::render_scene:
 […]
 for each channel_name in display_list do
    CHANNELS.displayChannel(channel name);
  end do
end proc

/// channel's rendering
procedure CHANNELS::displayChannel(channel_name):
      channels_list.lock();
      channel = channels_list.getChannel(channel_name);
      channel.display();
      channels_list.unlock();
end proc

/// channel's rendering
procedure CHANNEL::display:
      lockPicture();
      GRAPHICS.displayChannel(this.id,this.buffer);
      unlockPicture();
end proc
```

```
/// channel's display
procedure GRAPHICS::displayChannel(id,picture) :
      channel = getChannel(id);
      channel.updateTexture(picture);
end proc
```

We have four modules interacting here:

GRAPHICS: graphical engine

CHANNELS: video channels

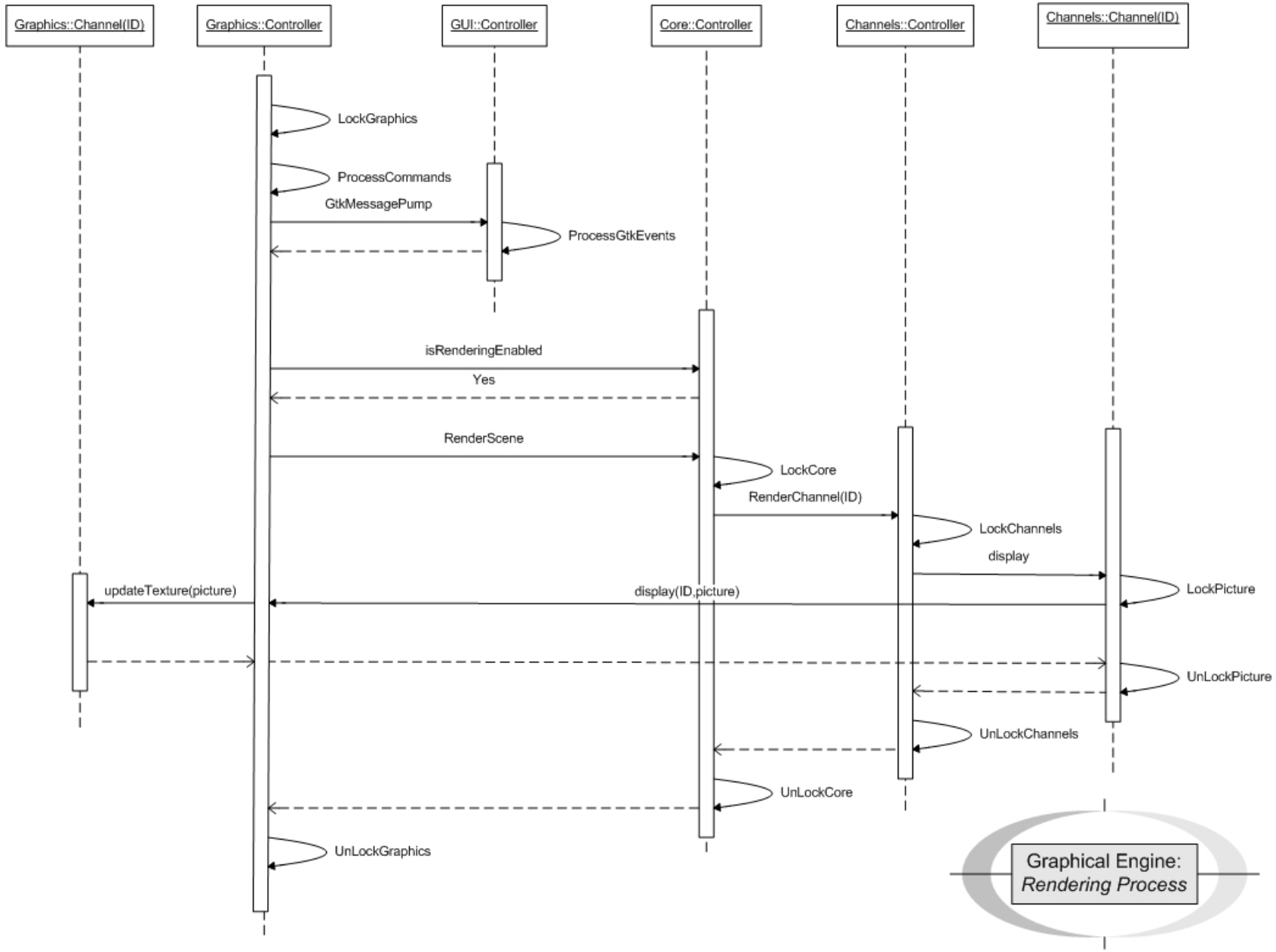
GUI: GTK GUI management

CORE: handle session information and link modules together.

First, the GRAPHICS ask the core to render the scene. The CORE will first render everything related to a Multivision (multiple cameras on the screen, cameras information, menus, etc.). Then, the CORE will loop through the list of channels ready to be displayed (registered as ready) and ask the CHANNELS module to display them. The CHANNELS module will find each corresponding channel and ask it to display its buffer. The channel will ask the GRAPHICS module to update its picture data. The GRAPHICS module will find the corresponding graphics channel and update the texture.

We see in this algorithm that each module is handling its own mutex and data. The GRAPHICS module doesn't directly access to the channel's buffer.

A modification of the above algorithm would be to record the channel's desire to be displayed directly in the GRAPHICS. The GRAPHICS module would then find the corresponding graphics channels and directly ask them to be displayed. Moreover, this allows the addition of decorators and other extra widgets to be displayed at the exact same time as we know which channel to target and when. The purpose is to progressively get rid of the CORE module and only have the other modules to interact together.



OpenGL Game Mode

The current graphical engine is still relying on a GTK layer to handle the GUI (windows, buttons, etc.). This forces OpenGL to run in a windowed mode (even if it is fullscreen). Currently, the GLUT fullscreen mode is basically made by a window covering the entire screen with off-screen borders. Moreover, the graphical engine loop runs the GTK message pump for every loop so that GTK messages are processed at the same speed as the graphical engine.

The downside of this dual GTK/OpenGL design is that OpenGL capabilities are limited due to the required interaction with the operating system in window mode. Indeed, GLUT windowed mode will have to process all Windows or Linux messages.

In OpenGL there is a mode, called Game Mode, which creates a direct link between the graphical card and the screen. This mode is mostly used by game makers. This mode is way faster than the windowed mode as OpenGL/GLUT disables all the window management and other related features, only focusing on rendering. This mode is really interesting but it prevents the use of a GTK-like GUI as windows cannot be drawn on screen and interacts with the OS anymore. Using such mode requires the programmer to design a complete OpenGL GUI which can take a lot of time.