ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y SISTEMAS DE TELECOMUNICACIÓN

PROYECTO FIN DE GRADO

TÍTULO: Creación de un software implementado sobre GPU para simulación de propagación de ondas y fluidos en diferentes entornos

AUTOR: David González Rodríguez

TITULACIÓN: Grado en Ingeniería de Sonido e Imagen

TUTOR
Dr. Szirmay-Kalos László

UNIVERSIDAD: Budapest University of Technology and Economics
CENTRO: Faculty of Electrical Engineering and Informatics
PAÍS: Hungría

Fecha de lectura:

Calificación:

El Coordinador de Movilidad,
Bachelor’s Thesis

Creation of software implemented on GPU for simulation of propagation of waves and fluids in different environments

David González Rodríguez

Degree in sound and image engineering

May 2017
GRATEFULNESS

For my family, for giving me the opportunity to make this happen. To my brother, to be there when I needed it and when not. To my mum, who never letting me ever leave and to fight for me every day.

For my tutor László and the entire computer graphics department of BME University, especially József.

For all my Erasmus friends, for helping me to see life from another point of view. Especially to Julio, Ana and Silvia.

For my student friends from Spain, who have passed this difficult and long path with me. Thanks, Roberto, María, Elena, Marta, Marlem, Alfredo and especially Laura.

To end, for my friends, my Community, Aarón, Ginés, Catalin and Dylan. Thank you for being there always, in good and bad. For putting up with me while I was in the degree, I will thank you forever.
Resumen

La investigación en los campos de física e ingeniería implica la necesidad de utilizar conocimientos muy técnicos y específicos. Esto genera el requisito de utilizar un gran número de tecnologías especiales para poder realizar los cálculos necesarios y poder obtener un resultado aproximado del experimento o investigación.

Para reproducir los fenómenos físicos que ocurren en la naturaleza se necesitan estudiar una gran cantidad de datos, provocando así la necesidad de realizar muchos cálculos para obtener los resultados finales buscados. Una vez obtenidos estos resultados, se debe decidir si se consideran favorables o desfavorables y si el estudio alcanza los objetivos esperados. En caso de descartar los resultados, los recursos y el tiempo invertido en la investigación pueden suponer una gran pérdida para el proyecto. Una manera de evitar este problema es la utilización de simuladores para predecir los resultados aproximados de la investigación. Los simuladores estudian una gran cantidad de información, lo que puede provocar que, aunque se ahoren recursos y dinero, aumente notablemente el tiempo necesario para obtener dichos resultados.

Por estas razones, el presente Proyecto de Fin de Grado tiene como objetivo crear dos programas que permitan la simulación de ondas electromagnéticas, de calor y de sonido dentro de una habitación. El conocimiento físico aplicado parte de dos ideas principales: la propagación del calor y la propagación de las ondas en un determinado entorno. Además, se describe la forma de creación de un software para simular fluidos en función de los datos necesarios a representar. Actualmente hay muchos programas de simulación basados en los procesos físicos elegidos en este proyecto, pero la gran mayoría de ellos utiliza la unidad central de procesamiento (CPU) como procesador principal para realizar los cálculos requeridos, por lo que el tiempo invertido para obtener los resultados puede ser muy grande.

Por ello, se ha decidido utilizar tecnologías novedosas para la creación de los softwares, consiguiendo así reducir el tiempo de procesamiento de los datos y el tiempo necesario para representar los resultados. Estas tecnologías son el estándar OpenGL y la arquitectura CUDA, ambas utilizan la unidad de procesamiento gráfico (GPU) en lugar de utilizar la CPU. El estándar OpenGL se utiliza para generar gráficos en 2D y 3D utilizando la tarjeta gráfica a partir de geometrías sencillas. Algunos ejemplos de uso de esta tecnología son la realidad virtual, simuladores de vuelo o desarrollo de videojuegos. La arquitectura CUDA es una tecnología desarrollada y utilizada por la empresa NVIDIA sobre sus tarjetas gráficas, la cual permite el uso de miles de hilos trabajando en paralelo para realizar los cálculos de una forma mucho más rápida. Otra ventaja de esta tecnología es la utilización de memoria compartida por los hilos, consiguiendo así un menor tiempo de acceso de memoria entre ellos.
Abstract

Engineering and physical research involves the need to use very specific knowledge, which causes the requirement to have a large number of special technologies to perform all expected and necessary calculations. Physical phenomena need a lot of data, which causes a lot of calculations to obtain the final results sought. Once these results have been obtained, it is decided whether they are considered favourable or unfavourable and if the study achieves the expected objectives. In case the results are discarded, the time and resources invested in the research can be a great loss for this. Therefore, a good way to avoid this is to use simulators that can predict the approximate results of the research. These simulations can take a great deal of time to research which causes a slowdown in this, despite the resources it saves.

Because of it, this Bachelor’s Thesis aims to create two programs that allow the simulation of electromagnetic, heat and sound waves. The applied physical knowledge starts from two main ideas, the propagation of heat inside a room and the propagation of waves in a certain environment. Also, the implementation of a fluid program will be commented in the project, but not implemented. There are currently many simulation programs based on the physical processes chosen in the present project, but most of them use CPU processing, which causes the large amounts of time already commented on to obtain the final results.

Therefore, it has been decided to use novel technologies that allow the creation of fast software to perform the necessary calculations, thus achieving a reduction in the waiting time to obtain the simulated results. These technologies are the OpenGL standard and the CUDA architecture, both used by the Graphics Processor Unit (GPU).

The OpenGL standard is used in different applications to generate graphs in 2D and 3D using the graphics card from simple geometries. Some examples of use are virtual reality, flight simulations or video game development. The CUDA architecture is a technology created by the company NVIDIA and allows the use of thousands of threads to calculate the needs provided by the program or the application. Another great advantage of this technology is the shared memory that it offers to the threads, obtaining a shorter access time to memory.
Contents index

Contenido

Resumen ......................................................................................................................... 1
Abstract .......................................................................................................................... 3

1. Introduction ................................................................................................................ 11
   1.1. Objectives ........................................................................................................... 11
   1.2. Memory Structure .............................................................................................. 12

2. Knowledge and technology used ............................................................................ 15
   2.1. Physics ................................................................................................................ 15
       2.1.1. Heat .............................................................................................................. 15
       2.1.2. Wave ........................................................................................................... 16
       2.1.3. Fluids .......................................................................................................... 17
   2.2. Programming ...................................................................................................... 19
       2.2.1. CUDA ........................................................................................................... 20
       2.2.2. OpenGL ...................................................................................................... 22
       2.2.3. Context switch ............................................................................................ 26

3. Physics calculations ................................................................................................. 27
   3.1. Heat equation ...................................................................................................... 27
   3.2. Wave equation .................................................................................................... 29
   3.3. Fluid equations ................................................................................................... 32

4. Software ................................................................................................................... 35
   4.1. Heat program ....................................................................................................... 35
       4.1.1. First implementation .................................................................................... 35
       4.2.1. Second implementation .............................................................................. 40
   4.2. Wave program .................................................................................................... 47
       4.2.1. First implementation .................................................................................... 47
       4.2.2. Second implementation .............................................................................. 49
   4.3. Fluid program ..................................................................................................... 56

5. Budget ....................................................................................................................... 59

6. Conclusion ................................................................................................................ 61

7. Future Improvements ............................................................................................... 63

8. Bibliographic references .......................................................................................... 65
Figures index

Knowledge and technology used
Figure 1. Representation of the heat flow from different source. ........................................... 15
Figure 2. Representation of a set of small masses connected between them. [WIKI17].......... 16
Figure 3. Flow chart of the process do by the programs. ............................................................... 19
Figure 4. Representation of the distribution of threads in CUDA [ZEL17]................................. 20
Figure 5. Distribution of the memory for CUDA architecture [RON17]................................. 21
Figure 6. Example of code, in language C, between standard code and parallel code. .......... 22
Figure 7. Typical graphics system [GAR17]. ........................................................................... 23
Figure 8. Example of a surface made by triangles in OpenGL [TOM17]................................. 24
Figure 9. Example of an OpenGL scene with some source lights [GEE17]. ......................... 25
Figure 10. Comparison example between OpenGL and DirectX 11 [YOU17]. .................... 25
Figure 11. Representation of a context switch. ...................................................................... 26

Physics calculations
Figure 12. Representation of heat flow. ..................................................................................... 27
Figure 13. Pulse of a wave. ........................................................................................................ 29

Software
Figure 14. Function Diffuse used to calculate the temperature flows. .................................. 36
Figure 15. Flow chart of the process do by the CUDA auxiliary heat program..................... 36
Figure 16. Loop 100 (left) and 2000 (right) of the example heat room. ............................... 37
Figure 17. Loop 5000 (left) and 10000 (right) of the example heat room............................. 37
Figure 18. Code lines which create the texture in OpenGL. ................................................... 38
Figure 19. Representation of the room with 5 sources (left) and the white texture (right). .... 38
Figure 20. Flow chart of the process do by the OpenGL auxiliary heat program.................. 39
Figure 21. Function to save memory for CUDA in the GPU. ................................................. 40
Figure 22. Function use to copy the memory from the host to the device. ............................ 40
Figure 23. New Function to make the call function for the Diffuse one................................. 41
Figure 24. New Diffuse function in the final heat program.................................................... 41
Figure 25. Function Source, used by add_Source to create a new heat origin ..................... 42
Figure 26. Function to map the PBO to allow CUDA access that resource ......................... 42
Figure 27. Function to get the pointer of the mapped resource. ........................................... 42
Figure 28. Function to unmapped the resource used by CUDA. ........................................... 43
Figure 29. Flow chart of the final heat program. .................................................................... 43
Figure 30. State of the room example at the beginning of the program execution (left) and some time later (right). ................................................................. 44
Figure 31. Example room for different times of execution of the program. Earlier (left) and some time later (right). .................................................................................. 44
Figure 32. State of the room example at the beginning of the program execution (left) and some time later (right). .................................................................................. 45
Figure 33. Example room for different times of execution of the program. Earlier (left) and some time later (right). .................................................................................. 45
Figure 34. Problems generated by entering a value of $g = 20$ without sources (left) with sources (right). .............................................................................................. 46
Figure 35. Problems generated by entering a value of $g = 20$ and $dt=1$ without sources (left) with sources (right). .................................................................................. 46
Figure 36. Function wave1 to calculate the middle velocity and middle height. ....................... 47
Figure 37. Function wave2 to calculate the final height and final velocity. ............................. 48
Figure 38. Flow chart of the process do by the CUDA program in the auxiliary wave program. 48
Figure 39. Loop 80 (left) and loop 2500 (right) of the example room. ............................. 49
Figure 40. Loop 5600 (left) and loop 9300 (right) of the example room. ............................. 49
Figure 41. Functions to save memory for CUDA in the GPU. ............................. 50
Figure 42. Function use to copy the memory from the host to the device. ............................. 50
Figure 43. Changes to make the call function for the Waves ones. ........................................ 51
Figure 44. New Wave1 function in the final wave program. ........................................ 51
Figure 45. New Wave2 function in the final wave program. ........................................ 51
Figure 46. Flow chart of the final wave program. ........................................ 52
Figure 47. Function Height, used by gen_Height to create a new wave pulse. ............................. 53
Figure 48. State of the wave at the beginning of the program execution (left) and some time later (right). ........................................ 53
Figure 49. Wave for different times of execution of the program. Earlier (left) and some time later (right). ........................................ 54
Figure 50. State of the wave at the beginning of the program execution (left) and some time later (right). ........................................ 54
Figure 51. Wave for different times of execution of the program. Earlier (left) and some time later (right). ........................................ 55
Figure 52. Problems generated by entering a value of dt = 0.001 at the beginning of the program execution (left) and some time later (right). ........................................ 55
Figure 53. Example of velocity (left) and display variable (right) textures [GPG]. ..................... 56
Figure 54. Pictures taken from an animation of Eulerian fluid dynamics [GPG]. ..................... 57
Figure 55. Behaviour of a fluid with an external force. ........................................ 57
Tables index

Table 1. Range of temperature, colours and value for RGB for the auxiliary program. ............ 35
Table 2. Range of temperature, colours and value for RGB for them for the final program. .... 40
Table 3. Range of height, colours and value of RGB for the final program. .............................. 50
Table 4. Comparative prices in function of the GPU................................................................. 59
1. Introduction

Nowadays, thousands of investigations are carried out related to a large number of fields, such as medicine, science, engineering... All of them are based on the search for a final result based on initial data. In most of them, physical or biological concepts are used to achieve these results, forcing the need to use a great knowledge. These concepts usually carry a great amount of calculations and tests to get to the final objective, being able to prolong the investigation for days or months. An efficient way to reduce the number of required tests as well as the great need for calculations are the simulations.

A simulation allows us to see in a fast and simple way the results that can be expected with a great precision. Thanks to this, the results obtained can be estimated before spending a lot of time and resources in the research. Although the simulations save a great deal of time, if it is necessary to use a lot of data for the realization of the simulation calculations, it can take a long time to complete. Therefore, the present project focuses on the creation of a wave simulator based on the use of novel technologies that allow a fast calculation and representation of the results of the simulation.

These technologies are the architecture CUDA for the GPU and the Multilanguage OpenGL to create the software which are explained in the section 2.1. Thanks to them, the estimated time for the simulations performed will be greatly reduced by saving large amounts of resources and time.

1.1. Objectives

The project is based on the creation of simulators for different physical phenomena, so the objectives of this are the following:

1. Creation of a simulator of heat propagation in a room with different heat sources in its interior. The GPU will be used to calculate the data and draw it on the display. The CPU will be used to write the results in the screen of commands and make modifications requested by the user in the data to avoid faults in the accesses inside and outside the GPU.

2. Creation of a simulator of wave propagation of a room with different disturbances in its interior. The GPU will be used to calculate the data and draw it on the display. The CPU will be used to write the results in the screen of commands and make modifications requested by the user in the data to avoid faults in the accesses inside and outside the GPU.
3. Obtain the necessary equations and propose a program of representation for these in the field of fluids. In this case, the GPU and the CPU would perform the same functions as in the previous programs.

1.2. Memory Structure
To understand how the programs work and in which technologies and physical knowledge are based, the present project is divided into five parts: Knowledge and technology used, physics calculations, the final designed software, a little conclusion and the bibliographic references. These parts are explained below.

1. **Knowledge and technology used:** In this part, the physical processes will be superficially detailed to explain the mathematical and physical foundations on which the programs are based. These physical facts are the propagation of the waves, differentiating the propagation of the heat with the rest of them and the Navier-Stokes equation for the fluids field. In addition, this part contains information related to the technology used in the project to understand the innovation used. As already mentioned, this technology is CUDA and OpenGL and the context switch to mixed them.

2. **Physics calculations:** This section corresponds to the explanation of the mathematical and physical concepts used in the programs. It is made a further development of the formulas used and how to obtain the necessary values for the final program. In the case of the transmission of heat, the heat equation and the transmission of heat in an environment will be taken. In the case of the propagation of waves in an environment the starting point will be the general wave equation. In the case of fluids, the velocity field and the forced involved in its perturbation will be the main equations.

3. **The software:** This section will describe the final programs. The auxiliary programs developed for each of the technologies will be commented on, and the final program that includes both auxiliary programs will be explained later. In the case of software created to develop the heat program will be described in more depth. In the other cases, they will not be exposed with so much information, because they will be developed from the heat program and will try to avoid the repetition of information.

4. **Budget:** Some examples of prices of the necessary components will be shown in this section.

5. **Conclusion:** The main objectives will be compared with the final objectives of the project, commenting on problems encountered or improvements achieved in the programs.
6. **Future improvements:** In this part, some improvements for the project will be commented. They have not been made due to lack of time or complexity in the proposition.

7. **Bibliographic references:** In the last section will refer to all the documentation used for the development of the project, including web pages, presentations and electronic books or PDF files or similar.
2. Knowledge and technology used

As discussed in the introduction, this memory is based on the creation of simulations programs related with the heat flow and the wave flow in two dimensions. Therefore, the areas covered by this project are the physics to understand these natural processes and the programming behind the software that allows its simulation at a great speed and the sample of results in a simple way to understand.

2.1. Physics

To generate the simulators, all the useful equations must be implemented and mixed in each program to create all the values at each point/pixel for each instant of time. In this project, the simulators will be prepared for two dimensions, so the equation only will study two directions of the heat flow and the wave flow.

2.1.1. Heat

To create the heat program, the heat flow must be study. In a room without heat sources inside, and assuming it is completely isolated, the temperature in each point in the room must be the same. If a heat source is introduced into it, the temperature will begin to change starting from the points closest to this heat source. Depending on the average temperature of the room and the power of the heat source, temperature will flow more quickly or less. Also, the number of heat source inside the room must be studied. The greater number of sources the greater the heat propagation within the room. An example of the heat flow from the source can be seen in figure 1.

![Figure 1. Representation of the heat flow from different source.](image-url)
In figure 1 the red lines represent the heat flow and the orange arrows represent the direction of the flow. So, in every instant of time the heat flows inside the room and modify the temperature in every point. If the sources do not have any type of condition and exist for an indefinite time, at a certain time will reach a thermal balance in the room. Between the initial time and this time, the temperature of the room will change for each instant of time.

So, to get this heat flow, the heat and the heat flow equations must be used. These equations correspond with [1] and [2] respectively.

\[
\frac{\partial T}{\partial t} = -k \left( \frac{\partial f_x}{\partial x} + \frac{\partial f_y}{\partial y} \right) = -k \nabla \cdot f \tag{1}
\]

\[
f = (f_x, f_y) = -\rho \left( \frac{\partial T}{\partial x}, \frac{\partial T}{\partial y} \right) = -\rho \nabla T \tag{2}
\]

Where ‘\(T\)’ correspond with the temperature, ‘\(k\)’ is the heat capacity and ‘\(f\)’ is the heat flow for the heat equation and ‘\(f\)’ correspond with the flow, ‘\(T\)’ with the temperature and ‘\(\rho\)’ thermal conductivity, heat capacity the for the flow equation.

Starting from these equations the final value will be obtained for each point and for each instant of time. In the section 3.1., the formulas [1] and [2] will be used to calculate the temperature value in every point.

2.1.2. Wave

For the wave simulator, the wave propagation must be study. To obtain the propagation of waves, a brief example will be used. The figure 2 [WIKI17] represents a set of small masses connected between them by springs of rigidity ‘\(k\)’ and separation ‘\(h\).

\[
\begin{array}{ccc}
\text{(m)} & \text{k} & \text{(m)} \\
\text{u(x)} & \text{u(x+h)} & \text{u(x+2h)} \\
\end{array}
\]

*Figure 2. Representation of a set of small masses connected between them. [WIKI17]*

Where ‘\(u(x)\)’ measures the displacement of the mass in the position ‘\(x\)’.

The equation of a force corresponds with the equation [3].

\[
F = m \times a 
\tag{3}
\]

Where ‘\(m\)’ is the mass of the object who is suffering the force, and ‘\(a\)’ the acceleration of it.

So, if a force is applied to the mass in the position \(x+h\), assuming a single direction of propagation, its value is:

\[
F = m \times \frac{\partial^2}{\partial t^2} u(x + h, t)
\]
This force is given by the Hooke Law, where the result is:

\[ F_h = F_{x+2h} + F_x = k \cdot [u(x + 2h, t) - u(x + h, t)] k \cdot [u(x, t) - u(x + h, t)] \]

If the last two equations are mixed, the outcome is:

\[ m \cdot \frac{\partial^2 u (x + h, t)}{\partial t^2} = k \cdot [u(x + 2h, t) - u(x + h, t)] k \cdot [u(x, t) - u(x + h, t)] \]

And assuming several masses equal to N, the total length ‘L’ is N*h, the total mass ‘M’ is N*m and the rigidity of the set ‘K’ is k/N. Modifying these changes in the last formula, it will be obtained:

\[ \frac{\partial^2 u (x + h, t)}{\partial t^2} = \frac{(K \cdot L)^2}{M} \cdot \frac{u(x + 2h, t) - 2 \cdot u(x + h, t) + u(x, t)}{h^2} \]

If the number of masses is assumed equal to infinity, and the distance between them, ‘h’, is equal to zero, the final expression corresponds with the equation [4].

\[ \frac{\partial^2 u}{\partial t^2} = \frac{(K \cdot L)^2}{M} \cdot \frac{\partial^2 u(x,t)}{\partial x^2} \] [4]

Where \( \frac{K \cdot L^2}{M} \) is the square of the wave speed, ‘c’. Making this change, the wave equation for one dimension [5] is obtained.

\[ \frac{\partial^2 u}{\partial t^2} = c^2 \cdot \frac{\partial^2 u(x,t)}{\partial x^2} \] [5]

The same equation for two dimensions correspond with the equation [6].

\[ \frac{\partial^2 u}{\partial t^2} = c^2 \cdot \left( \frac{\partial^2 u(x,y,t)}{\partial x^2} + \frac{\partial^2 u(x,y,t)}{\partial y^2} \right) \] [6]

This equation will be used in the section 3.2. to obtained the wave displacement in each point for a specific time.

2.1.3. Fluids

In this last part, the fluids dynamics equations must be discussed. The model of fluids is given by the Navier-Stokes equation, which correspond with non-lineal partial derivatives of the fluid movement. To obtain this equation, the principle of conservation of mechanics and principle of conservation thermodynamics to a fluid volume must be used.

To make the explanation, the equations of velocity and pressure of a fluid will be the starting point. These are [7] and [8] formulas.

\[ \vec{v} = \vec{v}(\vec{r}, t) \] [7]

\[ p = p(\vec{r}, t) \] [8]
If a fluid element of unit volume at a specific time, an earlier time instance for it must be the expression [9].

\[ \vec{v}(\vec{r}, t) = \vec{v}(\vec{r} - \vec{\nu} \cdot dt, t - dt) + \frac{\vec{F}}{\rho} \cdot dt \]  

Where \( \frac{\vec{F}}{\rho} \) is the acceleration change corresponding to the variation of the force ‘F’ divided by the mass of this element of fluid volume ‘\( \rho \)’.

Obtaining the substantial derivative of the velocity from equation [9]:

\[ \frac{\vec{v}(\vec{r}, t) - \vec{v}(\vec{r} - \vec{\nu} \cdot dt, t - dt)}{dt} = \frac{\vec{F}}{\rho} \]

The total pressure force corresponds with the pressure difference. This equality can be seen in the equation [10].

\[ \vec{F}_p = (\frac{\partial p}{\partial x}, \frac{\partial p}{\partial y}, \frac{\partial p}{\partial z}) = -\vec{\nabla}p \]

The viscosity ‘\( \nu \)’ affects the movement on the fluids, changing its damping property. Its equation is the number [11], where describe the relation between its force and the velocity.

\[ \vec{F}_v = \nu \cdot (\frac{\partial^2 \vec{v}}{\partial x^2} + \frac{\partial^2 \vec{v}}{\partial y^2} + \frac{\partial^2 \vec{v}}{\partial z^2}) = \nu \cdot \nabla^2 \vec{v} \]

In addition, a force external to the system must be assumed creating the acceleration.

Using all this information the Navier-Stokes equation can be implemented to obtain the velocity of a fluid. This equation corresponds with the number [12].

\[ \rho \cdot \frac{\vec{v}(\vec{r}, t) - \vec{v}(\vec{r} - \vec{\nu} \cdot dt, t - dt)}{dt} = -\vec{\nabla}p + \nu \cdot \nabla^2 \vec{v} + \vec{F}_{ext} \]
2.2. Programming

To implement the programs, a pair of new technology will be used. These technologies are CUDA and openGL. The first one is a NVIDIA parallel computing architecture which use the graphic card to calculate the operations needed in the program. The principal advantage of this technology is the enormous power that has the GPU (Graphic Processing Unit) to increment the efficiency of the system. The second one is a standard specification which defines a API to create 2D and 3D graphics applications. This specification also use the GPU to calculate and draw the graphics requested by the program.

These technologies used the C/C++ language programming, but could not use in the same code, because of this, a convert switch must be implemented. A convert switch is the switching of the CPU or GPU between different thread or processing to another. In this case, the graphic card has a lot of cores to perform the calculations at the same time, so we change all this block of threads between CUDA and OpenGL depending on what the program needs.

The structure of the programs is the same in all cases. First, the GPU, using the architecture CUDA, will calculate all the information related with the physics (in the heat program will calculate the temperature in every point and in the wave program will calculate the height of the wave in every point too) starting from the initial data. Once the data have been calculated, the graphic card will draw the information for every pixel using OpenGL, so, the context switch must be used. These steps will be looped by the program for each increment of time until it reaches the established limit. The figure 3 shows a flow chart of the process do by the software’s.

![Figure 3. Flow chart of the process do by the programs.](image)
2.2.1. CUDA

As discussed above, CUDA architecture will be used in the program, so it must be explained.

CUDA (Compute Unified Device Architecture) is a specific architecture used by NVIDIA graphics cards. It releases date was the fourteenth of June in 2007 and can be supported by all the GPU of the series G8X or superior, including GeForce, Quadro, ION and the line Tesla. This architecture use the GPU (device) to execute the functions call in the CPU (host), reducing the time of calculation that needs the devices to finish the operations.

This reduction of execution time is not only due to the fast access to memory that has the GPU. A GPU using CUDA has two major advantages:

1. Number and threads distribution: The architecture CUDA works using thousands of threads at the same time. To do this in a simple way, CUDA distribute all the threads in blocks and, in turn, these are distributed in grids. The figure 4 [ZEL17] displays a better representation of the distribution.

![Figure 4. Representation of the distribution of threads in CUDA](image)

As the figure 4 shows, every grid has a different number of blocks divided in one, two or three dimension, like a matrix of blocks. The same happens with the blocks, they are formed by a big amount of threads where are distributed in one, two or three dimension, like a matrix of threads. These blocks and thread are identified by indices which corresponds with their positions in the matrix. The number of blocks and threads are selected by the user.
So, for a specific function call by the CPU but execute in the GPU, one grid (compose by blocks and these by threads) will compute the function using all its threads at the same time, executing the same code for each one.

2. **Different types of memory share by the architecture:** CUDA uses a specific type of share/global memory. It should be defined the different types of memory that CUDA has.
   1. **Memory for every thread:** Each thread has a specific local memory to use. This memory is maintained until the thread finish the operations.
   2. **Memory for every block:** Each block has a shared memory to which any thread of the block can access it. This memory exists as long as the threads within the block are computing.
   3. **Global Memory:** All the threads from every block and every grid can access it.

The figure 5 [RON17] display an example of this explanation.

![Figure 5. Distribution of the memory for CUDA architecture [RON17].](image)

Also, CUDA has two more types of memory, but they are read only, *constant* and *texture* memory.

A thread has different memories to access during its execution time until it finishes its calculations.
A little example to show the difference between a standard code and a parallel code (in language C) can be seen in the figure 6.

![CUDA C](image)

Figure 6. Example of code, in language C, between standard code and parallel code.

In this example, the first code makes the operation 1,048,576 (4096*256) times, one by one. On the other hand, the second code make the same amount of operations, but instead of making it one by one it makes all the calculations at the same time, reducing the execution time greatly.

For these reasons, CUDA has been chosen to develop the project, getting the large number of operations needed in a much shorter time than if the CPU were used.

2.2.2. OpenGL

OpenGL is the other technology on which this project is based. This technology is a standard that defines a Multilanguage API used on different platforms. This technology generates graphs in 2 and 3 dimensions from basic geometries like points, lines and triangles. Some examples of this technology are the flight simulations, virtual reality, videogame development, Scientific representation and visualization of information. For the project the most important ones are the last two.

Originally, OpenGL was a C-type graphics library used only by native Silicon Graphics INC (SGI) machines that developed it. At that time, OpenGL was known as GL (Graphics Library). With the passage of time it was considered the possibility of being used in different platforms, ensuring its extensibility and its permanence. From this moment, it is known as OpenGL.

OpenGL runs on a graphical system, which receives some data as inputs. These inputs are the changes that have occurred in the scene and they must be modified in the screen, like a change in the main camera, the movement of the light source or a physical change inside of it. The graphic system processes this information and transform it into
the final data that be displayed on the screen, these are the outputs. To make the modifications, the graphic system is made up of five parts: the processor, the memory, the frame buffer, the Look Up Table and the converter between digital data and analogic data. An example of a typical graphic system can be seen in the figure 7 [GAR17].

![Typical graphics system](GAR17)

*Figure 7. Typical graphics system [GAR17].*

The main features of each element will now be discussed.

1. **Processor (CPU):** Administrator of the program. Makes the connexion between the different element. Its main function is to access the memory when requested and perform the necessary operations. Usually are several CPUs working in parallel to achieve a greater performance and speed in the system.

2. **Memory:** Element Which stores the information used for the CPU as well as the data obtained after the calculations applied.

3. **Frame Buffer:** Area of the memory where stores all the information of what must be drawn. In this zone, OpenGL will write and then send to the screen to draw it.

4. **Look Up Table (LUT):** This table has the value of each colour palette stored in it. These colours will be used to draw the pixels. This is not typical nowadays because this is only for index colours, now it normal use the RGB type.

5. **Converter Digital/Analogic (D/A):** The information that will be shown on the screen is in analogic type. The data stored in the memory is digital. So, an element must be used to transform this data to a specific information in analogic type. This is only needed in case the outputs must be on analogic type.
To render a 2D scene in OpenGL the input must be processed in a specific order to obtain the final result. The starting point is the scene in the modelling space, also called reference state. Then, the necessary changes will be made. These steps are five: vectorization, modelling transformation, normalize device space, clipping and viewport transform and rasterization.

1. **Vectorization:** Curves are approximated by polylines and areas by polygons (using the two ears theorem for example). This is necessary because in OpenGL, as already mentioned before, it can only process points, polylines and triangles. An example of a surface made by triangles can be seen in the figure 8 [TOM17].

![Figure 8. Example of a surface made by triangles in OpenGL [TOM17].](image)

2. **Modelling transformation:** This step changes the modelling space coordinate to world coordinates. The normal transformations are rotation, scaling and translation. These changes are made by a matrix modification of the position of the points, polyline or triangle vertex.

3. **Normalize device space:** The value of the device space goes between the coordinate (-1, -1) and the coordinate (1, 1). So, the world coordinates must be normalized to these values.

4. **Clipping and Viewport transform:** If the polyline or the polygons are off the viewport, they must be clipped. This involves removing the parts that are outside the range of the screen. (An example of method to now the clipping of a polygon is the Sutherland-Hodgeman polygon clipping). Also, in this step maybe some transformations must be done. The transformation here are the same as the modelling transformation: scaling, rotation and translation. In this case, the only difference is the matrix used to apply the changes due to the new coordinates of the space used. To finish, this part also change the window coordinates to the physical screen dimensions.
5. **Rasterization:** It is the process by which the polylines and polygons are transformed into pixels of a screen or pixels of a bit map. When these geometries are drawn, their representation must have the illusion of the real geometry drawing a few pixels. For example, it impossible to draw a perfect line, but it can be simulated a virtual line that closely resembles reality.

In a scene, also must consider the physical processes within this, as can be the reflection/refraction of the light within it. Two examples of a 3D scenes correspond with the figure 9 [GEE17].

![Figure 9. Example of an OpenGL scene with some source lights [GEE17].](image)

There are many languages and ways to represent scenes on a screen, such as direct X, but in this project, will only use OpenGL. An example of the differences between OpenGL and Direct X 11 can be seen in the figure 10.

![Figure 10. Comparison example between OpenGL and DirectX 11 [YOU17].](image)
2.2.3. Context switch

The last technology to speak about is the context switch. The context switch is the chosen option to get the communication between the two technologies discussed above. There are other possibilities, such as creating a middleware between both systems to achieve communication, but are less efficient for the purpose of the project. Therefore, a context switch is a method of communication between two programs, languages, etc. Which allow the sharing of a section of the device memory. This allows fast, inter-process information delivery and access by both parties. In this case, the two processes used in the final program correspond with the physics part (the CUDA ones) and the display part (the OpenGL part). An example of what its expected can be seen in the figure 11.

So, every time the program needs to draw something in the viewport, and it is in the CUDA calculations part, a process change must be made using the context switch. It is very important not to make these accesses to memory at the same time, this could cause a concurrency problem and generate a failure in the program or a problem in writing / reading from memory.
3. **Physics calculations**

This part explains the physical formulas used and the calculations made with them to design the software. It is divided in two parts, the first one for the heat equation and the second one for the wave equation.

3.1. **Heat equation**

To calculate the temperature in each point, the formula of the temperature as of the heat flow must be used. In the figure 12 a representation of the heat flow can be seen. It shows the variation of the flow for a specific point (in the program each pixel will correspond with a point of the room), where the flow changes depending on the and the old values of it and the variation of the axes x and axes y. Therefore, both axes must be studied.

![Figure 12. Representation of heat flow.](image)

To draw in the software the exchange of heat in a room, the heat equation for two dimensions [1] commented in the section 2.1.1. will be used:

\[
\frac{\partial T}{\partial t} = -k \left( \frac{\partial f_x}{\partial x} + \frac{\partial f_y}{\partial y} \right) = -k \nabla \cdot f
\]  

[1]

Where ‘\( T \)’ correspond with the temperature, ‘\( k \)’ is the thermal conductivity, and ‘\( f \)’ is the heat flow.

As already commented in section 2.1.1., the heat flow generated by the temperature difference [2] must be used:

\[
f = (f_x, f_y) = -\rho \left( \frac{\partial T}{\partial x}, \frac{\partial T}{\partial y} \right) = -\rho \nabla T
\]  

[2]
If these equations are mix, the expression [13] is obtained:

$$\frac{\partial T}{\partial t} = -g \left( \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) = g \nabla^2 T$$  \[13\]

Where ‘g’ is equal to ‘k’ multiplied by ‘ρ’.

To solve this equation the approximation [14] must be used.

$$\frac{\partial T_{x,y}}{\partial x,y} \approx \frac{T(x+dx,y) - T(x,y)}{2dx,dy}$$  \[14\]

Developing this equation for the second derivative and in the case of x:

$$\frac{\partial^2 T}{\partial x^2} = \frac{\frac{\partial T}{\partial x}(x + dx, y) - \frac{\partial T}{\partial x}(x - dx, y)}{dx} = \frac{T(x + dx, y) - T(x, y) - T(x, y) - T(x - dx, y)}{dx^2}$$

In the case of axes y, it has the same solution:

$$\frac{\partial^2 T}{\partial y^2} \approx \frac{T(x, y + dy) - 2T(x, y) + T(x, y - dy)}{dy^2}$$

So, to calculate the temperature variation at a given point between to instant of time corresponds to equation [15].

$$\frac{T_{x,y}(t+dt) - T(t)}{dt} = g \left( \frac{T(x+dx,y)-2T(x,y)+T(x-dx,y)}{dx^2} + \frac{T(x,xy+dy)-2T(x,y)+T(x,xy-dy)}{dy^2} \right)$$  \[15\]

And if the $T_{x,y}(t+dt)$ is cleared, the temperature value for the next time instant is obtained [16].

$$T_{x,y}(t + dt) = T(x, y) + dt \times g \left( \frac{T(x+dx,y)-2T(x,y)+T(x-dx,y)}{dx^2} + \frac{T(x,xy+dy)-2T(x,y)+T(x,xy-dy)}{dy^2} \right)$$  \[16\]
The expression [16] can be used to obtain the value of the temperature in a certain point and in a specific time, knowing the value of it and the adjacent temperature values in the last instant of time.

3.2. Wave equation

In this section, the wave equation will be calculated. The start point is the study of a pulse of a wave. The figure 13 shows an example of this.

![Wave diagram](image)

**Figure 13. Pulse of a wave.**

The important part for the wave simulator is the displacement of the wave in every point. The wave displacement can be seen in the figure 13 as the function ‘$u(x, t)$’, where ‘x’ is the position of the wave in axes X and ‘t’ the time. This software draw the displacement in two dimension, so the axes Y must also be studied.

So, to draw the displacement of a wave in the space, the wave equation [6] will be the starting point.

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u = c^2 \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = a$$

[6]

Where ‘$u$’ corresponds with the wave displacement, ‘$c$’ is the wave speed in the environment and ‘$a$’ is the wave acceleration.

To solve the wave equation, the second derivative of the wave displacement must be used. The formula for axes x and y correspond with the equations [17] and [18] respectively.

$$\frac{\partial^2 u}{\partial x^2} \equiv \frac{u(x+dx,y,t)-2u(x,y,t)+u(x-dx,y,t)}{dx^2}$$

[17]

$$\frac{\partial^2 u}{\partial y^2} \equiv \frac{u(x,y+dy,t)-2u(x,y,t)+u(x,y-dy,t)}{dy^2}$$

[18]
The value shown by the program must be the displacement of the wave as a function of time, this equation is [19].

\[ u(t + dt) = u(t) + v(t + dt) \times dt \]  

[19]

And to solve this equation the formula for the wave speed in a certain time [20] must be used.

\[ v(t + dt) = v(t) + c^2 \nabla^2 u(t + dt) \times dt \]  

[20]

The acceleration is the derivative of velocity with respect to time, and the speed of the wave is the derivative of the wave displacement. These formulas can be seen in the equation [21] and [22].

\[ a = \frac{\partial v}{\partial t} = c^2 \nabla^2 u \]  

[21]

\[ \frac{\partial u}{\partial t} = v \]  

[22]

By relating both equations, the equation [23] can be obtained:

\[ \frac{\partial [u,v]}{\partial t} = [c^2 \nabla^2 u, v] \]  

[23]

Combining the equations [6], [17], [18] and [21], the first derivative of the wave velocity with respect to time can be obtained:

\[ \frac{\partial v}{\partial t} = c^2 \times u(x + dx, y, t) - 2u(x, y, t) + u(x - dx, y, t) + \frac{u(x, y + dy, t) - 2u(x, y, t) + u(x, y - dy, t)}{dy^2} \]

Solving this combination, the wave speed in a specific time can be obtained:

\[ \frac{v(t + dt) - v(t)}{dt} = c^2 \times u(x + dx, y, t) - 2u(x, y, t) + u(x - dx, y, t) + \frac{u(x, y + dy, t) - 2u(x, y, t) + u(x, y - dy, t)}{dy^2} \]

\[ v(t + dt) = v(t) + c^2 \times u(x + dx, y, t) - 2u(x, y, t) + u(x - dx, y, t) + \frac{u(x, y + dy, t) - 2u(x, y, t) + u(x, y - dy, t)}{dy^2} \times dt \]
If the equation [22] is also solved, the value for the wave displacement in a specific time can be obtained too:

\[ v(t) = \frac{u(t + dt) - u(t)}{dt} \]

\[ u(t + dt) = v(t) \cdot dt + u(t) \]

To finish, the expression of the wave speed in a specific time 't' can be obtained using the equation [23]:

\[ v(t) = v(t+dt) - c^2 \cdot dt \left[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right] \]

And using the equations [6], [19] and [20] in this expression, the wave displacement for a particular time 't+dt' can be obtained the final formula [24]:

\[
\begin{align*}
    u(x, y, t + dt) &= u(x, y, t) + dt \cdot v(x, y, t + dt) - (c \cdot dt)^2 \cdot \\
           &\quad \left[ \frac{u(x+dx,y,t)-2u(x,y,t)+u(x-dx,y,t)}{dx^2} + \frac{u(x,y+dy,t)-2u(x,y,t)+u(x,y-dy,t)}{dy^2} \right] \\
\end{align*}
\]

[24]

There is a problem resolving the wave equation in programming, an infinite sum can appear causing an unstable value for the wave displacement, where never will stop and it value increased indefinitely.

Because of it, the system must be linear to secure that it doesn’t occur in the program and the representation will be correct.

The best solution for this complication is the Verlet integration. This integration used the expression [25] to solve the problem.

\[ x(t + dt) = x(t) + f \left( \frac{x(t)+xe(t+dt)}{2}, t + \frac{dt}{2} \right) * dt \]

[25]

Where \( xe(t + dt) \) corresponds with the formula [26].

\[ xe(t + dt) = x(t) + f(x, t) * dt \]

[26]

And \( f(x, t) \) must be the formula [27].

\[ \frac{dx}{dt} = f(x, t) \]

[27]
So, the solution for the equation [26], using the formula [27] is the next one:

\[ x(t + dt) = x(t) + f \left( \frac{x(t) + x(t) + f(x, t) \cdot dt}{2}, t + \frac{dt}{2} \right) \cdot dt = \]

\[ = x(t) + f \left( x(t) + \frac{dx}{2}, t + \frac{dt}{2} \right) \cdot dt \]

It follows that to obtain the value of \( x \) at a point and at a given time \( t + dt \), we must calculate the value of \( x \) at time \( t + dt \) before and use it to calculate the required value.

3.3. Fluid equations

To obtained the fluid equations, the Eulerian approach must be used in the fluids Navier-Stokes equation [12]. This method simplifies the infinite elements of a function in a finite set of values, allowing to make the necessary calculations based only on this group of data. This approximation corresponds with the equation [28].

\[ f(\vec{r}) \approx \sum_{i=0}^{N} f_i \cdot B_i(\vec{r}) \]  

[28]

Where ‘\( f \)’ is the function that is going to be approximated, ‘\( B_i(\vec{r}) \)’ are the pre-defined basis functions and ‘\( f_i \)’ are the coefficients that describe ‘\( f \)’. The particularity of the Eulerian method is to maintain constant the sample points (\( r \)) and to vary only the coefficients ‘\( f_i \)’ as a function of time.

Mixing this concept with the Navier-Stokes equation [12], the value for operators in it can be approximated to [29] and [30]. These equations are solved for the two-dimensional space.

\[ \nabla \vec{p} \approx \left( \frac{p^{x, y} - p^{x-1, y}}{2*dx}, \frac{p^{x, y+1} - p^{x, y-1}}{2*dy} \right) \]  

[29]

\[ \nabla^2 \vec{v} = \left( \frac{v^{x+1, y} - 2*v^{x, y} + v^{x-1, y}}{\left(dx\right)^2} + \frac{v^{x+1, y} - 2*v^{x, y} + v^{x-1, y}}{\left(dy\right)^2} \right) \]  

[30]

If the fluid is incompressible, the velocity field is divergence free because of the mass conservation. The velocity and is approximated operator can be seen in the formulas [31] and [32].

\[ \nabla \cdot \vec{v} = \left( \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} \right) \]  

[31]

\[ \nabla^2 \vec{v} \approx \left( \frac{v^{x+1, y} - 2*v^{x, y} + v^{x-1, y}}{2*dx}, \frac{v^{x, y+1} - 2*v^{x, y} + v^{x, y-1}}{2*dy} \right) \]  

[32]
Using all these approximations, the final Navier-Stokes equation [33] can be obtained for an advancing time of length ‘dt’.

\[ \vec{v}(\vec{r}, t) = \vec{v}(\vec{r} - \vec{v} \ast dt, t - dt) + \frac{F_{ext}}{\rho} \ast dt + \frac{\nu dt}{\rho} \vec{V} \vec{v} - \frac{dt}{\rho} \ast \vec{Vp} \]  \[ 33 \]

There some consideration that must be applied to implement the fluids formula in a program.

The advection must be considered to initialize the velocity field, studying the actual position and the last one of the fluid. This consideration corresponds with the formula [34].

\[ \vec{w}_1(\vec{r}) = \vec{v}(\vec{r} - \vec{v} \ast dt, t - dt) \]  \[ 34 \]

The equation [35] can be obtained damping the velocity field to a diffusion term.

\[ \vec{w}_2 = \vec{w}_1 + \frac{\nu dt}{\rho} \vec{V} \vec{w}_1 \]  \[ 35 \]

Another consideration, is the external force field. This force accelerates the fluid velocity at each point of the system. The equation [36] is the result of relating the velocity field with the external field the equation.

\[ \vec{w}_3 = \vec{w}_2 + \frac{dt}{\rho} \ast \vec{F}_{ext} \]  \[ 36 \]

Using the operators in equations [35] and [36], a sparse linear system of equations can be achieved.

The las consideration is the pressure field. This field must be updated the velocity field to get the equation [37].

\[ \vec{v}(t) = \vec{w}_3 - \frac{dt}{\rho} \ast \vec{Vp} \]  \[ 37 \]

Supposing that the final velocity field must be divergence free, the final result will be:

\[ 0 = \vec{V}(\vec{w}_3 - \frac{dt}{\rho} \ast \vec{Vp}) = \vec{Vw}_3 - \frac{dt}{\rho} \vec{Vp} \]
4. Software

In this section, the realization of the simulators will be commented step by step. Both programs use CUDA and OpenGL but, due to the complexity of these technologies, two auxiliary test programs will be performed, one for each technic. These auxiliary programs only will be made for the heat program, for the wave simulator they are not necessary because it will be created from the heat program, which must already work.

Once the auxiliary programs have been implemented, the heat software can be performed using the context switch discussed in section 2.2.3. This is the most difficult part of the project since to connect both technologies must be done with great care without forgetting any step, since both use the GPU as a calculation unit and can cause great problems when implementing a program.

4.1. Heat program

With the calculated temperature values, the heat fluctuation could be draw. To transform them to a texture of OpenGL, each pixel will be considered a variation of x and y, which will have a specific temperature value.

It is non-viable to store all the information related with the temperature to draw it, so it will be drawn at the same time the changes are calculated.

To do this, two arrays of dimension roomHeight*roomWidth are used to store the new and old values respectively.

The scale of colours selected for the pixels is divided into seven parts. Each part has an initial colour and a final colour and all temperature values within the range are a variation between both. This temperature division, its limits and the value of RGB for each colour could be seen in the Table 1.

<table>
<thead>
<tr>
<th>Range of Temperature</th>
<th>Scale of Colours (R, G, B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-study pixels</td>
<td>Black (0,0,0)</td>
</tr>
<tr>
<td>-273°C → -200°C</td>
<td>Black (0,0,0) → Blue (0,0,1)</td>
</tr>
<tr>
<td>-200°C → -100°C</td>
<td>Blue (0,0,1) → Green (0,1,0)</td>
</tr>
<tr>
<td>-100°C → 0°C</td>
<td>Green (0,1,0) → Yellow (1,1,0)</td>
</tr>
<tr>
<td>0°C → 50°C</td>
<td>Yellow (1,1,0) → Orange (1,0.5,0)</td>
</tr>
<tr>
<td>50°C → 150°C</td>
<td>Orange (1,0.5,0) → Red (1,0,0)</td>
</tr>
<tr>
<td>More than 150°C</td>
<td>Red (1,0,0)</td>
</tr>
</tbody>
</table>

4.1.1. First implementation

Before the final OpenGL and CUDA program will be created, the heat equation must be programmed to secure that it works in separated parts. Therefore, a simple software was created to probe it using the CUDA architecture and other simple OpenGL program to ensure that the representation of heat through a texture works.
The auxiliary openGL software generates a heat array with differences temperatures in each point (the temperature is selected by the user) and a boundary condition array (which determines the boundary condition for each point). Then, the program allocates these arrays in the GPU memory and start computing the temperature flows using a block of threads of N columns and N rows (where N is the height/width of the room). The code related to the function that calculates these values corresponds with the figure 14.

```
void Diffuse(float* o, float* n, float* b, float g, float dx, float dy, float dt, int N)
{
    int x = threadIdx.x, y = blockIdx.x;
    float dTx2 = (T(o, N, x + 1, y) - 2 * T(o, N, x, y) + T(o, N, x - 1, y))/(dx*dx);
    float dTy2 = (T(o, N, x, y+1) - 2 * T(o, N, x, y) + T(o, N, x, y-1))/ (dy*dy);
    n[y*N + x] = (T(o, N, x, y) == 0) ? T(o, N, x, y) + g*(dTx2 + dTy2)*dt : T(o, N, x, y);
}
```

Figure 14. Function Diffuse used to calculate the temperature flows.

Where the parameters have the following meanings. ‘o’ is the array that stores the old values, ‘n’ is the array that stores the new values, ‘b’ is the boundary condition, ‘g’ is the value of \( k \) plus \( \rho \), ‘dx’ and ‘dy’ is the variation of x and y (in the program is always 1, corresponding for each pixel where x is the horizontal axes and y is the vertical axes), ‘dt’ is the variation of time (its value determines the resolution of the calculations) and N is the height/width of the room.

In every loop, the new values are calculated by the program and print to a file with extension ‘.tga’. The figure 15 shows a flow chart related with this implementation.

Figure 15. Flow chart of the process do by the CUDA auxiliary heat program.
A video can be made with the generated images. Some photograms of an example video can be seen in the figures 16 and 17.

![Figure 16. Loop 100 (left) and 2000 (right) of the example heat room.](image)

![Figure 17. Loop 5000 (left) and 10000 (right) of the example heat room.](image)

The auxiliary openGL program starts by asking the user the size of the window (height and width) and average temperature of the room. After that, the program will generate a window with the data obtained by the user and draw with a texture in it. This texture will be filled with random values created from the value entered by the user for the average temperature. If the window will be clicked by the user, the program will create a heat source in that position, asking before this the value of that source. Also, when the user press the button ‘space’, the software fill in the texture with white colour and then draw the entire window with it, simulating the onset of the heat propagation (this is the part do in CUDA).
The important part of this auxiliary software is the creation of textures in OpenGL, which help reduce processing time for the final program. The code lines that create the texture correspond with the figure 18.

```
glGenTextures(1, &textureID);
glBindTexture(GL_TEXTURE_2D, textureID);
TexImage2D(GL_TEXTURE_2D, 0, GL_RGB, roomWidth, roomHeight, 0, GL_RGB, GL_FLOAT, texDat);
TexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
TexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
Enable(GL_TEXTURE_2D);
BindTexture(GL_TEXTURE_2D, 0);

//match projection to window resolution
MatrixMode(GL_PROJECTION);
gluOrtho2D(0, roomWidth, 0, roomHeight);
MatrixMode(GL_MODELVIEW);

//clear and draw quad with texture
ClearColor(GL_COLOR_BUFFER_BIT);
BindTexture(GL_TEXTURE_2D, textureID);
```

**Figure 18.** Code lines which create the texture in OpenGL.

As the figure 18 shows, to create the texture we must generate an identifier for this texture. After this, we must bind the name to a texture target. Then, we create the texture related with this identifier and the values that must be actualized in every frame. Before finishing with the creation of the texture, some parameters of this must be declared as the filters to use in case of visual problems with the texture. To finish with this part, the type of projection must be declared and we must place the texture in the correct position to see it in the entire window.

The figure 19 represents an example of a room filled in with random temperatures with some sources (after selecting their position in the software) and the white texture (when the user push the space).

**Figure 19.** Representation of the room with 5 sources (left) and the white texture (right).
A flow chart of the auxiliary software can be seen in the figure 20.

Figure 20. Flow chart of the process do by the OpenGL auxiliary heat program.
4.2.1 Second implementation

Once the auxiliary programs work, the final program is implemented from them using a context switch. To do this, the final program must use two files, ‘HeatSimulator_Kernel.cu’ for the CUDA part and ‘HeatSimulator.cpp’ for the OpenGL part. These files have almost the same lines of code as the auxiliary programs with some additions and modifications.

In the OpenGL file, there must be implemented the memory location for the variables that CUDA is going to use to calculate the values of the temperature in every pixel. These lines correspond with the figure 21.

![Figure 21. Function to save memory for CUDA in the GPU.](image)

In the figure 21, ‘Tgpu1’ is the array of the new values of temperature in the room, ‘Tgpu2’ corresponds with the array that store the old values of the temperature in the room. Both arrays will be used to calculate the heat flow in every loop. ‘Bgpu’ is the array that stores the boundary conditions of the room, to know if that pixel can be modified or not. To finish with variables, ‘Nb’ is the number of bytes that this function must allocate in the memory. Its value depends on the room height and width and the size of the variable that use the arrays (in this case are floats).

Once the memory has a specific section to store the necessary data, the software must copy the values of the temperature in every point and the boundary restrictions every time they change (For example when the user clicks with the left mouse button the window to create a source). This can be done with the functions that appear in the figure 22.

![Figure 22. Function use to copy the memory from the host to the device.](image)

After allocating the arrays in the memory and asked for the input values, the program draws the room and starts the simulation of the heat flow. To improve the visualization of the temperature in the room, the range of colours for the different temperatures has been modified. The new ranges can be seen in the Table 2.

| Table 2. Range of temperature, colours and value for RGB for them for the final program. |
|--------------------------------------|---------------------------------|
| Range of Temperature | Scale of Colours (R, G, B) |
| -50°C → 0°C | Blue (0,0,1) → Black (0,0,0) |
| 0°C → 50°C | Black (0,0,0) → Green (0,1,0) |
| 50°C → 100°C | Green (0,1,0) → Yellow (1,1,0) |
| 100°C → 150°C | Yellow (1,1,0) → Orange (1,0,5,0) |
| 150°C → 200°C | Orange (1,0,5,0) → Red (1,0,0) |
| More than 200°C | Red (1,0,0) |
Regarding the part of CUDA, the code has been perfected allowing the implementation of rectangular rooms, because the auxiliary program was only created to verify that the calculations were performed correctly.

The first modification is the call of the function that performs the calculations. In the auxiliary program, every pixel has a thread to get the temperature value, but with large rooms the program run out of memory. To fix this problem, the function call change to the way that appear in the figure 23.

```c
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid((int)(roomHeight / dimBlock.y) + 1, (int)(roomWidth / dimBlock.x) + 1);
Diffuse<<<dimGrid, dimBlock>>>(d_output, Tgpu1, Tgpu2, Bgpu, g, dx, dy, dt, roomWidth, roomHeight);
Diffuse << <dimGrid, dimBlock >> >(d_output, Tgpu1, Tgpu2, Bgpu, g, dx, dy, dt, roomWidth, roomHeight);
```

*Figure 23. New Function to make the call function for the Diffuse one.*

Also, has had to modify the function Diffuse, to get to realize the calculations inside a rectangular room. Before, having the same dimension with a single value was not necessary, but now both dimensions must be studied. In addition, when making the modification discussed above the way to obtain the value of the studied pixel has change. The new Diffuse function can be seen in the figure 24.

```c
void Diffuse(uint *d_output, float* o, float* n, float* b, float* g, float dx, float dy, float dt, uint w, uint h)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < w && y < h)
    {
        float4 color;

        float dTx2 = (T(o, w, h, x + 1, y) - 2 * T(o, w, h, x, y) + T(o, w, h, x - 1, y)) / (dx*dx);
        float dTy2 = (T(o, w, h, x, y + 1) - 2 * T(o, w, h, x, y) + T(o, w, h, x, y - 1)) / (dy*dy);
        n[y*w + x] = (T(b, w, h, x, y) == 0) ? T(o, w, h, x, y) + g*(dTx2 + dTy2)*dt : T(o, w, h, x, y);
        color = calculateRGB(n[y*w + x]);
        d_output[y*w + x] = rgbaFloatToInt(color);
    }
}
```

*Figure 24. New Diffuse function in the final heat program.*
The final program uses mostly the GPU. So, if a heat source wants to be created, it must be done in it. To get this, the function ‘add_Source’ must be call from the OpenGL file and then, CUDA must access to the memory and actualize the array for the new temperature with the new source. In the next calculation loop, the source will start to take effect. The function ‘add_Source’ is call by a CUDA kernel and its global function can be seen in the figure 25.

1. Once the arrays were allocated in the memory, a graphics resource must be mapped to allow CUDA to access it. This can be done with the function that shows the figure 26.

Every time the resource must need to be accessed (every loop to draw it in OpenGL), a pointer to that mapped resource must be obtain. The function in figure 27 allowed the program to do that. With this pointer, the program will be able to access the values of the resource and modify the texture in function of the values of them. Afterwards, the program could draw the texture in the viewport.

2. After that, the program will call the CUDA kernel to calculate the new values for the temperature using the function ‘Diffuse’ shown on the figure 14. When the software finishes with the calculation, the values in the array ‘Tgpu1’ will actualize automatically. Every time the program wants to access this data from the OpenGL file, a memory copy must be made between device and host because the read / write of the GPU must be very careful. A device function can also be performed in the CUDA file that allows its modification.
3. When the temperatures values are store in OpenGL, the resource must be unmapped to allows OpenGL use its information to actualize the texture and the draw it in the viewport. To unmapped the resource the function which appears in the figure 28 must be call.

```c
checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_pbo_resource, 0));
```

*Figure 28. Function to unmapped the resource used by CUDA.*

The figure 29 shows a flow chart that correspond with the final program.

*Figure 29. Flow chart of the final heat program.*
To end the explanation of the final heat program, two examples of heat flow for different values will be shown.

The first example corresponds to a room with an average temperature of twenty degrees. This room has 5 heat source. The figures 30 and figure 31 shows different moments of the room, arranged in order according to the elapsed time.

**Figure 30.** State of the room example at the beginning of the program execution (left) and some time later (right).

**Figure 31.** Example room for different times of execution of the program. Earlier (left) and some time later (right).

The red heat sources have two hundred degrees, the yellow ones have one hundred degrees and the blue one has minus fifty degrees.
The other example is a room with one hundred and fifty degrees as average room. This room only has one source in the center of it. The figures 32 and 33 shows different moments of the room, arranged in order according to the elapsed time.

**Figure 32.** State of the room example at the beginning of the program execution (left) and some time later (right).

**Figure 33.** Example room for different times of execution of the program. Earlier (left) and some time later (right).

While the program is running, if the user presses the right mouse button, the current temperature will be displayed for the cursor point.
The program is prepared to avoid entering large or small values by keyboard for the screen dimensions and for the average temperature of the room and the sources. But if the values for the variables involved in the equation of temperature calculation (‘dx’, ‘dy’, ‘g’ and ‘dt’) are modified, they can cause problems in the representation.

In the examples shown in the figures 30, 31, 32 y 33 the values used are the following: g=2, dx=dy=1 and dt=0.1. If these values are modified with larger numbers, problems appear as shown in Figures 34 and 35. Both examples have a middle temperature of two hundred Celsius degrees.

![Figure 34. Problems generated by entering a value of g = 20 without sources (left) with sources (right).](image)

![Figure 35. Problems generated by entering a value of g = 20 and dt=1 without sources (left) with sources (right).](image)
4.2. Wave program

Using the equation obtained in the section 3.2, the displacement of the wave can be drawn. Also, the final program will be used an OpenGL texture, considering each pixel for each point of the room.

As the heat program, it is non-viable to store all the information related with the wave propagation, so it will be drawn at the same time as the data are calculated. To do this, six arrays must be used, two for the new and old height, two for the new and old velocity and two for the middle height and middle velocity to implement the Verlet integration.

The value of a height equal to zero corresponds to the black colour and will increase until reaching the maximum height that will be green. The representation is simple because it only needs to corroborate that the program works correctly. In the final software will use a more colourful range of colours that allows better representation.

4.2.1. First implementation

Before the wave equation is implemented in the final program, it must be corroborated that the results obtained are as expected. In case of failures it will be simpler to correct the error in the auxiliary program than in the final one. The auxiliary program for representing textures in OpenGL will be the same as the one used in the heat program, so it will not be necessary to implement it.

The auxiliary CUDA program calculates the wave height for each point at a given instant of time by applying the formula of the wave equation [6]. So, it must use height for every point and the velocity in every point also. To obtain a value that not generate an infinite summation, the Verlet integration must be implemented, so it must divide the calculations in two parts, the first one obtains the middle velocity and middle height and the other obtain the final value for this point and for this specific time. The code related with these functions can be seen in the figures 36 and 37 respectively.

```c
void wave1(float* ou, float* ov, float* mu, float* mv, float c, float dx, float dy, float dt, int n)
{
    int x = threadIdx.x, y = blockIdx.x;
    float dT2a = T(ou, N, x + 1, y) + T(ou, N, x - 1, y) + T(ou, N, x, y + 1) + T(ou, N, x, y - 1) - 4 * T(ou, N, x, y);
    mv[y*N + x] = ov[y*N + x] + c*c*(dT2a)*(dt / 2);
    mu[y*N + x] = ov[y*N + x] + ov[y*N + x] * (ct / 2);
}
```

Figure 36. Function wave1 to calculate the middle velocity and middle height.

Where the parameters have the following meanings. ‘ou’ is the array that stores the old values for the height, ‘ov’ is the array that stores the old values for the velocity, ‘mu’ is the array that stores the middle values of the height, ‘mv’ is the array that stores the middle values of the velocity, ‘c’ is velocity of the wave in that environment, ‘dx’ and ‘dy’ is the variation of x and y (in the program is always 1, corresponding for each pixel where x is the horizontal axes and y is the vertical axes), ‘dt’ is the variation of time (its value determines the resolution of the calculations) and N is the height/width of the room.
Figure 37. Function wave2 to calculate the final height and final velocity.

Where the parameters have the same meaning as in wave1 but also use two array to store the final height and final velocity. They are ‘nu’ ‘nv’ respectively. As can be seen in figure 37, a dumping factor was used to make the program resemble reality by preventing an infinite sum of terms from being obtained and causing the program to oscillate. It is not a random value, it is an estimated value to obtain a constant energy during the existence of the wave. Also, instead of calculation dTx2 and dTy2, this program calculates directly the final second derivative of the height, ‘dT2u’.

In every loop, the new values are calculated by the program and print to a file with extension ‘.tga’. The figure 38 shows a flow chart related with this implementation.

Figure 38. Flow chart of the process do by the CUDA program in the auxiliary wave program.
A video can be made with the generated images. Some photograms of the video can be seen in figures 39 and 40.

**Figure 39.** Loop 80 (left) and loop 2500 (right) of the example room.

**Figure 40.** Loop 5600 (left) and loop 9300 (right) of the example room.

In a certain time, the program start to fluctuate and start incrementing the height of every point because the simulation not use the perfect dumping factor to get the exactly energy in every loop. In addition, the square pattern shown in Figures 39 and 40 is generated by the Verlet integration, which is really difficult to has a perfect resolution in the simulations, modifying with a little error the wave height on every pixel. This error is accumulative and depends on the parameters used.

4.2.2. Second implementation

Once the auxiliary program for the wave displacement work properly, the final program can be implemented. It must use two files as in the heat program, ‘WaveSimulator_Kernel.cu’ for the CUDA part and ‘WaveSimulator.cpp’ for the OpenGL part. These files are like the heat program files ones, but with some differences that are going to be explained.
In the OpenGL file, the memory location for the variables that CUDA is going to use will be the same, but adding the new arrays and putting the names correctly. These code lines can be seen in the figure 41.

![CUDA code lines](image)

**Figure 41.** Functions to save memory for CUDA in the GPU.

Where ‘Tgpu1u’ is the array of the old values of the height, ‘Tgpu2u’ is the array of the new value of the height, ‘Tgpu1v’ corresponds with the array of the old values of the velocity, ‘Tgpu2v’ is the array that stores the new values of the velocity, ‘Tgpumu’ corresponds with the array for the middle values of the height and ‘Tgpumv’ is the array that stores the middle values of the velocity. The last two arrays are only used to perform the Verlet equation.

After the memory has a specific section for the information, the program must copy the started values of the height and the velocity for every point to this array. To do this, the code lines that appear in the figure 42 must be called.

![CUDA code lines](image)

**Figure 42.** Function use to copy the memory from the host to the device.

The initial value is only necessary for the old arrays, the rest will be filled in each loop every time the calculations are performed.

When the program finish with the initialization of CUDA and allowing the arrays in the memory, it asks the user to enter the size of the room to study. After this, the program start the simulation, waiting until the user clicks with the left mouse button the window to make a force in the room that starts the wave.

In this case, the representation of the wave is draw only between two colours, one for the negative values and other for the positive values. The zero height is colour black. The range for the wave correspond with the Table 3.

**Table 3.** Range of height, colours and value of RGB for the final program.

<table>
<thead>
<tr>
<th>Range of Height</th>
<th>Scale of Colours (R, G, B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td>Green (0,1,0) → Red (0,0,0)</td>
</tr>
<tr>
<td>0</td>
<td>Black (0,0,0)</td>
</tr>
<tr>
<td>Greater than 0</td>
<td>Black (0,0,0) → Green (1,0,0)</td>
</tr>
</tbody>
</table>
Focusing on the part of CUDA and comparing it with the auxiliary program, the code has been improved to allow calculations in rectangular rooms as in the heat program. This software has the same problem as the heat program, large rooms make it run out of memory. Because of this, the changes that appear in the figure 43 must implement.

```c
dim3 dimblock/block_size, block_size);
dim3 dimGrid(int) [roomWidth / dimblock.x] + 1, (int)[roomHeight / dimblock.y] + 1;
Wave1 = dimGrid, dimBlock >> (Tgpu1, Tgpu1, Tgpu2, Tgpu2, c, dx, dy, dt, roomWidth, roomHeight);
Wave1 = dimGrid, dimBlock >> (d.output, Tgpu1, Tgpu1, Tgpu2, Tgpu2, c, dx, dy, dt, roomWidth, roomHeight);
Wave1 = dimGrid, dimBlock >> (Tgpu2, Tgpu2, Tgpu2, c, dx, dy, dt, roomWidth, roomHeight);
Wave1 = dimGrid, dimBlock >> (Tgpu2, Tgpu2, Tgpu2, c, dx, dy, dt, roomWidth, roomHeight);
```

**Figure 43. Changes to make the call function for the Waves ones.**

Apart from making this change, the functions Wave1 and Wave2 must be modified to get the displacement value of the wave at each point for rectangular rooms. The new functions correspond with the figures 44 and 45.

```c
void Wave1(float* u, float* v, float* h, float* c, float dx, float dy, float dt, uint u, uint h)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < && y < h)
    {
        float dt2u = T(ou, u, h, x + 1, y) + T(ou, u, h, x - 1, y) + T(ou, u, h, x, y + 1) + T(ou, u, h, x, y - 1) - 4 * T(ou, u, h, x, y);
        m[y + u + x] = u[y + u + x] + c^2*(dt2u)*(dt / 2);
        m[y + u + x] = u[y + u + x] + v[y + u + x] * (dt / 2);
    }
}
```

**Figure 44. New Wave1 function in the final wave program.**

```c
void Wave2(float* d.output, float* u, float* v, float* h, float* d, float dx, float dy, float dt, uint u, uint h)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < && y < h)
    {
        float4 color;
        float dt2u = T(ou, u, h, x + 1, y) + T(ou, u, h, x - 1, y) + T(ou, u, h, x, y + 1) + T(ou, u, h, x, y - 1) - 4 * T(ou, u, h, x, y);
        m[y + u + x] = u[y + u + x] + c^2*(dt2u)*(dt) * 0.9999;
        m[y + u + x] = u[y + u + x] + v[y + u + x] * (dt);
        color = calculateRGB(m[y + u + x]);
        d.output[y + u + x] = rgbaFloatToInt(color);
    }
}
```

**Figure 45. New Wave2 function in the final wave program.**

Every time the user clicks the room, the height of the point clicked will changed to a certain value that allows the wave to start propagating. The context switch in this program is the same as the heat program, so it will not be commented.
The figure 46 shows a flow chart that correspond with the final wave program.

*Figure 46. Flow chart of the final wave program.*
In the final wave program, the GPU calculate and draw all the information related with the program. So, if a pulse want to be created to start a wave, when the user clicks the window, must be done on the GPU memory. To get this, the program use the function ‘gen_Height’, which call the kernel CUDA function ‘height’. This function can be seen in the figure 47.

![Function Height](image)

*Figure 47. Function Height, used by gen_Height to create a new wave pulse.*

To end the explanation of the final wave program, two examples of wave propagation for different values will be shown. In addition, these examples will be used to display the problem of applying Verlet integration with different initial values.

The first example corresponds with a room with two wave pulse near the walls to see the reflection in it. The value of the initial variables are the next ones: c= 1000, dx=dy=1 ant dt = 0.00001. The figure 48 and 49 shows different states of the waves inside the room, arranged in order according to the elapsed time.

![WaveRoom](image)

*Figure 48. State of the wave at the beginning of the program execution (left) and some time later (right).*
Figure 49. Wave for different times of execution of the program. Earlier (left) and some time later (right).

The second example is a room with three waves. In this case, the c value was modified to show the differences between both examples. The new values are: c = 1000, dx=dy=1 and dt = 0.00001. The figure 50 and 51 shows different states of the waves inside the room, arranged in order according to the elapsed time.

Comparing both examples, we can appreciate how influential the parameters of the wave for the integration of Verlet can be.

Figure 50. State of the wave at the beginning of the program execution (left) and some time later (right).
In addition, the wave program is designed to avoid large or small values for user inputs. But if the code variables, that are used to get the wave displacement, are modified, they can cause problems in the representation. If the values are changed with large numbers, problems such as those shown in figure 52 appear.

**Figure 51.** Wave for different times of execution of the program. Earlier (left) and some time later (right).

**Figure 52.** Problems generated by entering a value of $dt = 0.001$ at the beginning of the program execution (left) and some time later (right).
4.3. Fluid program

The creation of a fluid program is a complex task because many values have to be studied when creating it. Also, depending on the type of representation needed for the program involves a big change in the code of this. Therefore, it has been decided to comment on the implementation of a general program that draws the main values of a fluid (Velocity, Eulerian fluids dynamics...).

The rendering program will be the same as the heat and wave software, because it draws a two-dimensional OpenGL texture. So if a magnitude of the fluid will be wanted to draw only a texture must be generated with these values and create a range of colours for that texture.

Some changes must be made in the CUDA part. In this case, three arrays must be used to store the speed and the different forces that affect the system. In each time loop, the equations for each of the magnitudes must be applied to obtain the new values of the arrays. Therefore, memory must be reserved for the six arrays that are needed in the program, three for the new values and three for the old values of each magnitude.

In addition, the necessary kernels must be created to compute this equation to the arrays, depending on the size of the container of the fluid. Some examples obtained from page 1484 of the GPGPU book, referenced in the bibliographic, can be seen in the figures 53 [GPG] and 54 [GPG].

![Figure 53. Example of velocity (left) and display variable (right) textures [GPG].](image-url)
Figure 54. Pictures taken from an animation of Eulerian fluid dynamics [GPG].

Figure 55. Behaviour of a fluid with an external force.

Figure 54 [GPG] shows a simulation of a wind tunnel (down) test and the expansion of a gas (above). An example of the behaviour of a fluid when an external force is applied is shown in Figure 55.
5. Budget

In this section, the project budget will be explained.

To carry out the project, a Graphics Processor Unit must be used. For this reason, the only expense necessary to develop the project is the price of the NVIDIA graphics card used. In the table 4 some prices for different cards can be seen.

Table 4. Comparative prices in function of the GPU.

<table>
<thead>
<tr>
<th>NVIDIA Graphic Card</th>
<th>Price (euros)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA TITAN Xp</td>
<td>1349€</td>
</tr>
<tr>
<td>GeForce GTX 1080 Ti</td>
<td>695€-1251€</td>
</tr>
<tr>
<td>GeForce GTX 1080</td>
<td>670€-1070€</td>
</tr>
<tr>
<td>GeForce GTX 1070</td>
<td>410€-810€</td>
</tr>
<tr>
<td>GeForce GTX 1060</td>
<td>215€-468€</td>
</tr>
<tr>
<td>GeForce GTX 1050 Ti</td>
<td>148€-274€</td>
</tr>
<tr>
<td>GeForce GTX 1050</td>
<td>119-182€</td>
</tr>
<tr>
<td>GeForce GT 740</td>
<td>104€-136€</td>
</tr>
<tr>
<td>GeForce GT 730</td>
<td>54€-234€</td>
</tr>
<tr>
<td>GeForce GT 720</td>
<td>67€-114€</td>
</tr>
<tr>
<td>GeForce GT 710</td>
<td>34€-73€</td>
</tr>
</tbody>
</table>

Table 4 shows some examples of graphics cards that use the CUDA architecture, but there is a great amount of possibility of choosing in the market of NVIDIA graphics cards.

It must be said that in order to use one of these graphics cards, it must be connected to an appropriate device that can work with it, such as a computer. In this section has not added the budget of the devices capable of supporting these cards because the market offers a multitude of products that can do it.
6. Conclusion

In this penultimate section will compare the initial objectives with the final results of each of the sections studied in the project.

All the objectives proposed in the project have been achieved with great efficiency. In the case of the heat software, the simulation was obtained with some improvement, such as the possibility of the user adding heat sources in the room at any time of the simulation or the possibility of obtaining the information of each point. In addition, it was possible to improve the implementation of the program by minimizing the need for CPU. Instead of using the CPU to display data and make changes made by the user, some functions were generated inside the CUDA file that allows to perform these processes in the GPU. These changes allow to save a great amount of calculations and time to the program and avoiding the need to copy information between the CPU and GPU and vice versa.

Focusing on the wave program, the integration of Verlet did not have the expected effect, because it generates some problems if the correct data is not entered or if very different data are chosen. On the other hand, we added the possibility that the user will interact with the simulation by adding pulses of wave in the room. In this program also the commented improvement for the heat program was implemented, extending the performance of the wave simulator.

Finally, the necessary bases have been left to carry out the fluid program in case one wants to program a software based on these calculations, only needing to apply the formula of the magnitude to be drawn. Improvement of the GPU can also be used in this software, because the fluid program would be based on the previous ones, which already have this new implementation.

In summary, all the main objectives of the project have been achieved along with some improvement within them. The proposed technologies have been applied with great efficiency, allowing very fast and effective simulators.
7. **Future Improvements**

Throughout the project, the steps taken to create the heat program and the wave program have been explained, as well as the basis for the creation of a fluid program. All this has great possibilities for improvement, some of them will be discussed in this section for each program.

The best improvement that can be applied for the programs is to get their representation in three dimensions. To do this, the necessary changes will be two. The first must be in the CUDA part, where the equations must be changed to add the dimension z. The second change must be the representation in OpenGL for three-dimensional space, which can be the most difficult and tricky part.

1. **Heat program:** this software is the most developed of all. It has a high resolution with respect to the representation of the temperature, but can be improved by applying the ideas commented below.

   - Creating a legend that is updated in each loop according to the maximum and minimum value that are shown on the screen.

   - Implement a function that allows to introduce obstacles in the room like columns, or large objects.

   - Improving the representation of rooms. Instead of generating only square and rectangular rooms, allow to creating rooms of different geometries, like curved walls or rooms with different areas.

   - Create a function that allows export / import a simulation together with its data so other programs will used them to work with.

   - Allow the introduction of windows and doors in the walls, adding different coefficients for these positions in the simulation. With this, the heat transmission inside the room could be studied more accurately.

The implementation of most of these upgrades would not cause great difficulty since only new functions should be created within the main program, without modifying the principal code (except for the upgrades of geometries of the room).
2. **Wave program**: This program simulates waves in a given environment. Its implementation was more complicated to achieve an acceptable resolution, because it depends on the input parameters. Some ideas to improve the program are as follows.

- Instead of applying the Verlet integration, use another method that allows to obtain a value of the wave with better resolution, avoiding the problems mentioned in section 4.2, generated by the implementation of this integration (Other methods that can be applied are Returning Euler, Crank-Nicolson method and Second Runge-Kutta).

- Use the Verlet integration, but creating an algorithm that relates the speed of the wave and the values of the variables used in the calculation of the wave height to avoid the errors explained in section 4.2.

- Implement a function that allows to introduce obstacles in the room like columns, or large objects.

- Create a function that allows export / import a simulation together with its data so other programs will used them to work with.

- Allow the introduction of variables such as the speed of the wave or the height of this when the screen is clicked.

The most difficult improvement to implement of all the commented, is the update of the integration of Verlet by another one that improves the resolution of the program. The rest of them only involve the creation of an external function that does not need major modification within the main code.

In the case of the introduction of obstacles in the room, an array should be added to store the boundary conditions of the pixels in the room. This would force a small modification in the main code, inhibiting the study of the propagation of the wave within this, as well as avoid drawing the pixels of the obstacle.

3. **Fluid program**: This software has not been completely implemented, the bases for its creation have been commented. Therefore, some ideas for improving it will be added.

- Implement a fluid program based on the equations obtained or the examples proposed. Improving the functionality of the program with improvements such as export / import data.

- Implement another usable method for the calculation and representation of fluids. (Lagrangian approach).
8. Bibliographic references


