ULB UNIVERSITÉ LIBRE DE BRUXELLES

INFO-F-420 Computational Geometry

Ray Tracing Problem

Silberwasser Elliot 000518397, Merian Emile 000518697

November 27, 2024

1 Introduction

This project aims to study the Ray Tracing problem using the javascript three module to visualize all the elements.

2 Formal definition of the Ray Tracing Problem

To correctly define the Ray Tracing problem, it is necessary to introduce some definitions.

2.1 Problem definition

A problem can be defined as a language. Each problem instance can be encoded using a finite word w on a finite alphabet Σ^*. Consequently, a language is an infinite set of finite words.

2.2 Decision problem definition

A decision problem is a problem where the answer is a binary yes or no. So, we decide the problem when we can give an answer for any given input.

The question to answer for the Ray Tracing Problem is the following:

Given an optical system (namely, a finite set of reflective or refractive objects), a light ray's initial position and direction, and some fixed point p, does the ray eventually reach p?[1]

Deciding a decidable problem comes down to knowing whether a word w ($\equiv$ an instance of the problem) belongs to the language L or not, where L represents the problem. It's called the Membership Problem.

$$w \in L?$$

To answer at this question, we need to consider three cases:

1. $w \in L$, w is a positive instance of $L \subseteq \Sigma^*$.
2. $w \in \Sigma^* \setminus L$, w is a negative instance ($\notin L$).
3. w is a finite words on Σ^* which does not correspond to the encoding rules for any instances of the problem, and therefore does not represent a valid instance of the problem considered.

2.3 Turing Machine definition

A Turing Machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$

- Q is the finite set of control states.
- Σ is the input alphabet not containing the blank symbol of the language.
- Γ is the tape alphabet.
- $\delta : Q \times \Gamma \to Q \times \Gamma$ is the transition function.
- $q_0 \in Q$ is the start state.
- $q_{accept} \in Q$ is the accepting state.
- $q_{reject} \in Q$ is the rejecting state, where $q_{accept} \neq q_{reject}$

A basic Turing Machine is therefore a machine with a tape which will contain the word w (the problem instance) to read, a reading head position and control states. The machine will move state depending on the input read by the tape head and the transition function which will determine the state to reach depending on the input.

When a Turing Machine stops in a state q_{accept}, then the word $w \in L$, otherwise, the word $w \notin L$. It is even sometimes possible for the Turing Machine to loop indefinitely on a word w.

Consequently, a Turing Machine can reject a word w in two cases:

1. If the terminal state is a q_{reject}.
2. When the Turing Machine loop on w and therefore that the Turing Machine does not stop on the input.

In this case, the Turing Machine is called a Turing-Recognizable language. It is the equivalent of saying that the problem ($\equiv$ the language) is undecidable because we do not know how to find an algorithm that solves all the instances of the problem in a finite number of steps has there is a possibility of the machine infinitely looping.

2.3.1 Turing-Decidable language

Principle: The Turing Machine stops on each input and never loops.

By definition:

A language $L \subseteq \Sigma^*$ is Turing-Decidable if a Turing Machine decides it.

In decidability, according to the Church's Theorem, the notion of algorithm is equivalent to the Turing Machine model. This makes it possible to use Turing Machines as a means of determining whether a problem is decidable or not and therefore whether there is an algorithm that runs in a finite number of steps that allows it to be decided.

2.4 Concept of reductions

Another important concept in decidability is the notion of reductions. This concept will allow us to prove or disprove the complexity and decidability of a problem based on its capacity to provide solutions to another problem of known decidability.

2.4.1 Reduction definition

A reduction is a way of converting one problem into another.

More formally:

A function $f : \Sigma^* \to \Sigma^*$ is a reduction from problem $A \subseteq \Sigma^*$ to problem $B \subseteq \Sigma^*$ if:

- f is a computable function, i.e there is an algorithm (termination is guaranteed - Church's Theorem) that give a finite word w, it will compute the finite word $f(w)$.
- $w \in A$ **if and only if** $f(w) \in B$. This point ensures that positive instances of the problem A are translated into positive instances of the problem B and same thing with the negative instances. This ensures the preservation of instances.

2.4.2 Reduction in decidability

Let A, B be two problems.

If problem A can be reduced to problem B, we can use the solution of problem B to solve problem A

This principle is very useful for showing the undecidability of a problem. To show that problem B is undecidable, we will show that problem A, which is a problem known as undecidable, can be reduced to problem B. Therefore, the problem B is also undecidable; otherwise, this contradicts the fact that the problem A is undecidable.

We can summarize the powerful of the reduction into two conclusions:

1. If there exists a reduction from the problem A to the problem B and we know that B is decidable, then the problem A must be decidable
2. Inversely, if there exists a reduction from the problem A to the problem B and the problem A is known undecidable the the problem B must me undecidable.

Note: All these definitions will be well used to show the decidability of the problems studied using this principle of reduction.

2.5 The Ray Tracing Problem - Formal definition

For this project, we will prove that we can solve any problem decidable by a reversible Turing machine using instances of the Ray Tracing problem. We will define precisely how to simulate this.

2.5.1 Definition of an instance

As mentioned above, an instance of the problem can be reduced to a word on a given alphabet. In our case, the alphabet is binary and is made up of the symbols $\{0, 1\}$.

The overall vision of the problem can be seen as a starting input corresponding to an entry point of the ray, which will then pass through the different reflective and refractive surfaces of the optical system. The Turing machine calculates after each passage through a surface the exit position by the deviation caused by the different surfaces.

In this way, the input will change on the tape depending on the modifications induced by the different surfaces of the optical system.

Therefore, an instance of the problem is defined by two binary fractions $U = 0.u_0.u_1.u_2...$ and $V = 0.v_0.v_1.v_2....$ These two words represent the coordinates (x, y) in decimal relative of a unit square.

We can apply two different operations on U and V.

1. **LeftMove** $\delta(q, c) = (q', c', L)$
2. **RightMove** $\delta(q, c) = (q', c', R)$

These two operations allow us to modify the encoding of the words U and V on the tape. The transformations U' and V' induced by the transition functions are the following.

For the LeftMove operation, we have the following transformations [1]

$$U' = 2\sum_{i=1}^{\infty} u_i/2^{i+1} = 2.(U - u_0/2)$$

$$V' = c'/2 + 1/2.\sum_{i=0}^{\infty} v_i/2^{i+1}$$
$$= c'/2 + v/2$$

For the RightMove operation, we have the following transformations [1]

$$U' = v_0/2 + c'/4 + 1/2.\sum_{i=1}^{\infty} u_i/2^{i+1}$$
$$= v_0/2 + c'/4 + (U - u0/2)/2$$
$$V' = 2.\sum_{i=1}^{\infty} v_i/2^{i+1} = 2(V - v_0/2)$$

These binary transformations will be transposed into geometry using reflective and refractive surfaces in order to represent and simulate an optical system.

2.5.2 Representation of an optical system

An optical system can be represented by an intertwining of "basic" boxes. By assembling these basic boxes together, it forms complex boxes which will represent the two transition functions LeftMove and RightMove mentioned above. Each complex box corresponds to one state of the Turing machine's finite control, and implements the transition function defined for that state.

Each complex box has a unit square through which the ray enters and one or two unit squares from which the ray exits. These unit squares are called the entrance windows and the exit windows, respectively. The tape is encoded by the (X, Y) coordinates of the ray relative to the unit square windows. We organize these complex boxes so that the whole system simulates the Turing machine. The ray enters the entrance window and exits out of one of the two exit windows depending on which state the Turing machine may enter next. The system then projects the ray onto the next complex box while preserving the coordinates of the ray relative to the window, thus simulating the transition of states defined on the Turing Machine. machine [1].

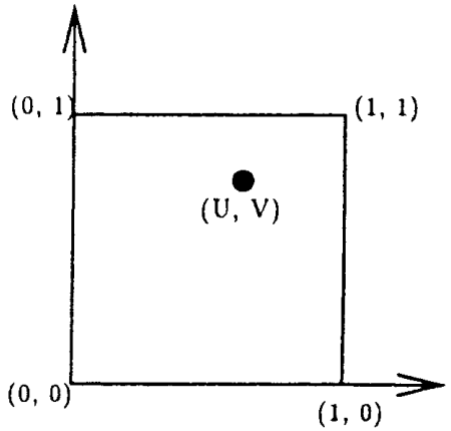


Fig 4. The representation of (U, V) in a unit square

There are 3 types of basic boxes to be able to represent binary transformations on U and V:

1. **Readout boxes** - The principle of this box is to simply redirect the ray to the left or right depending on the value of the word U given in input consequently according to the X coordinate of the first unit square by which the ray enters. This is made possible by the 90° inclination of two flat mirrors or prisms atop of the box such that if U is superior or equal to 0.5 (hence the first bit of U is 1), the ray will be redirected by the right mirror or prism to the right of the basic box, else to the left of the basic box by the left mirror or prism of the box.

There isn't exactly a Turing-Equivalent of this operation but it allows us to scout the values on the tape to know how to properly handle the shifting operation and especially know if we need to handle a bit overflow.

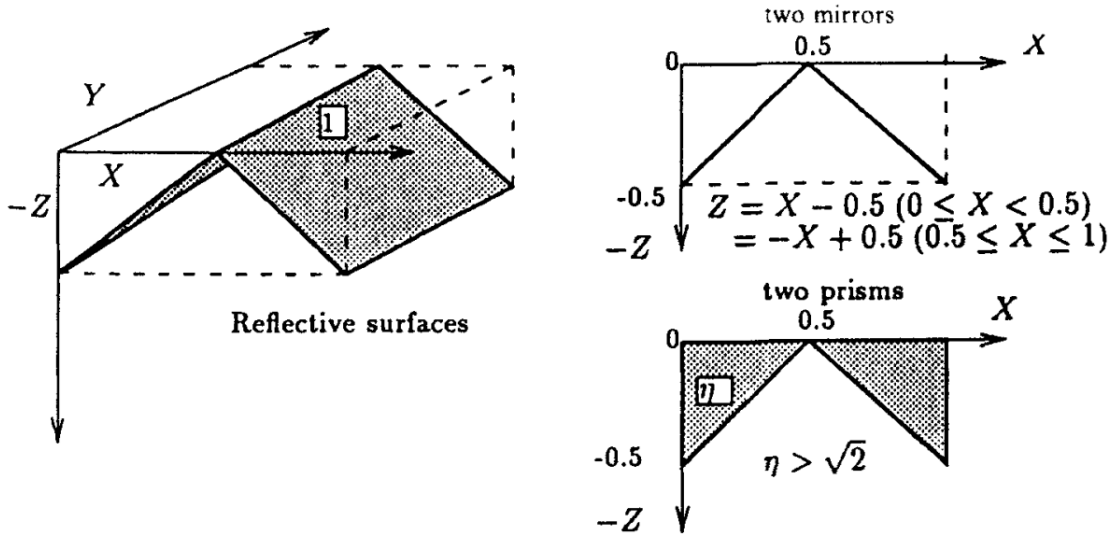


Fig. 6. Readout box

2. **Multiply2/Divide2 box** - The principle of this box is to redirect the ray into an exit coordinate such that one of its coordinate value is either double or half the value of the corresponding coordinate of the first unit square by which the ray enters. This is done by using two mirrors or lenses angled such that the distance of the ray to the focal line as shown on the figure below either halves or double depending on which side of the box the ray enters. By rotating the box from its horizontal axis parallel to the focal line, we can decide whether we modify the coordinate U or V and by reversing the box on any axis perpendicular to the focal line, we can decide to either multiply or divide that value.

   This is the Turing-equivalent of shifting the bits of U or V once to either the left or right within their allocated space on the tape, discarding bit overflow. To handle bit overflow, we will requires a beam turner that artificially adds the overflown bit from one coordinate value to the other. For example, to shift the tape head to the left, we need to multiply U by two and divide V by two, and adding a bit at the beginning of V if there was an overflow on U.

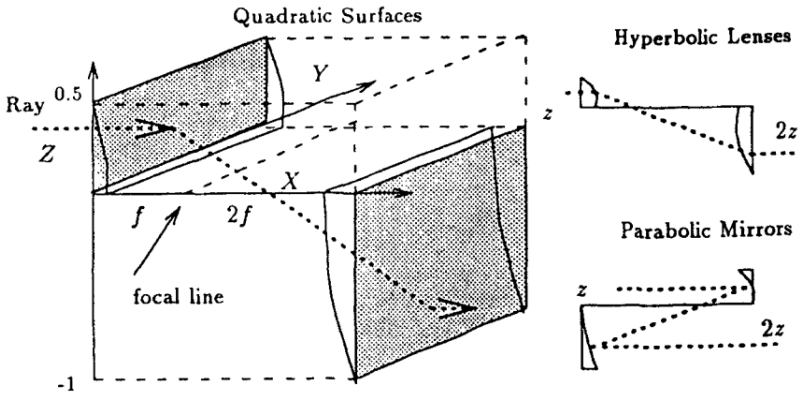
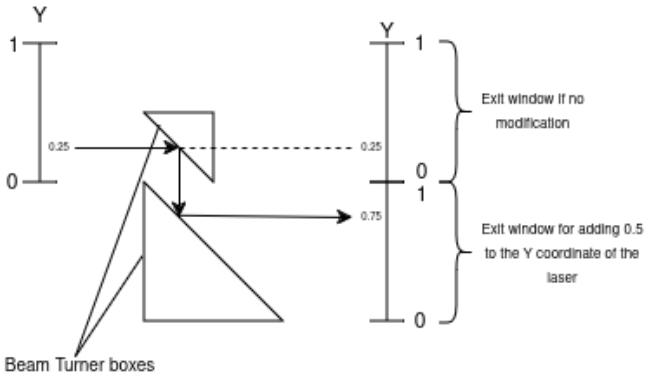


Fig. 7. Multiply2 (divide2) box using quadratic surfaces.

3. **Beam turner box** - The simplest box among the basic boxes, the beam turner will simply change the direction of the ray by $\pi/2$. It does so using a simple mirror or prism oriented at a $\pi/4$ angle. This box will be used to link boxes with each other and if used in pairs, can add or subtract a constant value to one of the coordinates of the ray.

   This will allow us to simulate the Turing-equivalent of replacing the value read by the tape head with a new value. Below is an example where we add 0.5 to the Y coordinate of the ray of light, indicated by the set of arrows.



These simple boxes will compose the complex boxes that forms the transition functions LeftMove and RightMove that we will now present.

1. **Implementing Left-Move** The complex box allowing to represent the state q of a Turing Machine with a Left-Move transition to a next state q' is described as follows :

   (a) The ray enters a readout box from its unit square called the entrance window of the state to check the value of U and redirect the ray accordingly to two distinct paths of operations.

   (b) The ray then enters a Multiply2 box that will divide U by two ignoring potential bit overflow. Now we will only consider the path chosen if $U >= 0.5$ in which we don't need to handle bit overflow since we the first bit of V will be the new value c' that replaces c.

   (c) The ray finally enters a two beam turner set of boxes depending on whether the written value c' is 0 or 1 and exits this state into the entrance window of the next state q' such that $U' = 2(U - u0/2)$ $V' = (V + c')/2$.

   (d) We can implement a similar structure on the chosen path if $U < 0.5$.

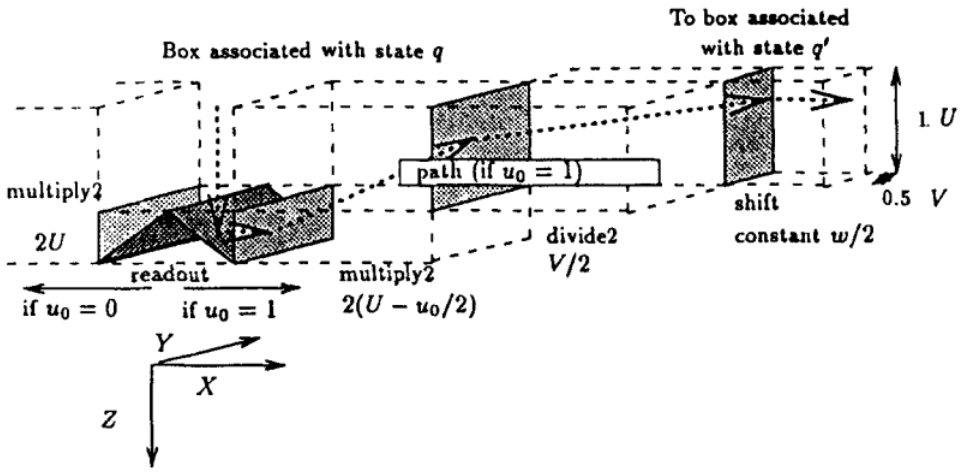


Fig. 8. Implementing $\delta(q, c) = (q', c', L)$.

2. **Implementing Right-Move** The complex box allowing to represent the state q of a Turing Machine with a Right-Move transition to a next state q' is described as follows :

   (a) The ray enters a Readout box from its unit square called the entrance window of the state to check the value of U and redirect the ray accordingly to two distinct paths of operations.

   (b) The ray then enters a Divide2 box that will divide U by two ignoring potential bit overflow. Now we will only consider the path chosen if $U >= 0.5$.

   (c) The ray then enters another Readout box where it will check the value of V and redirect the ray accordingly to two new distinct paths of operations. We will now only consider the the path chosen if $V >= 0.5$.

   (d) The ray finally enters a set of two beam turner boxes depending on whether the written value c' is 0 or 1 and exits this state into the entrance window of the next state q' such that $U' = v0/2 + c'/4 + (U - u0/2)/2$ and $V' = 2(V - v0/2)$.

   (e) We can implement a similar structure on the chosen paths if $U < 0.5$ and/or $V < 0.5$.

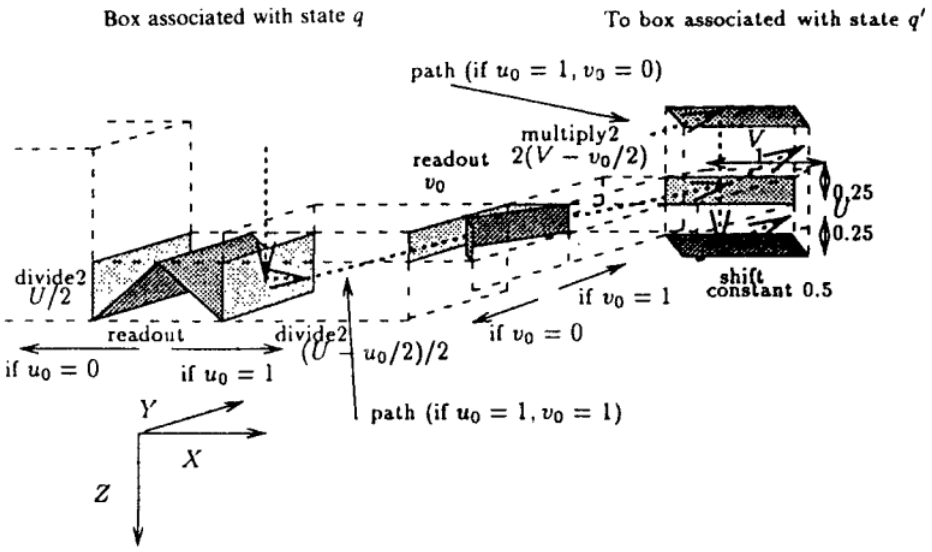


Fig. 9. Implementing $\delta(q, c) = (q', c', R)$.

2.5.3 Reaching a final state

We can simulate the initial transition to the first state of the machine by projecting the ray at the position that represent the input of the Turing Machine. In other words, we can simply aim the ray at the first complex box of our system. To simulate the transition to a final state, we simply designate a state box to be the final state. Therefore, if the ray doesn't reach this final box, the input coordinates are rejected, while if the ray does reach the final box, the input coordinates are accepted.

2.5.4 Undecidability of the optical system

We have now demonstrated the capability of the optical system to operate like a Turing Machine and simulate any reversible Turing machine where the number of complex boxes required is equal to the number of finite states of the Machine. Therefore the problem of predicting the exact trajectory of a light ray, otherwise called the Ray Tracing problem, can be as difficult as a problem of reaching a final state for an equivalent reversible Turing Machine. Since the latter is undecidable, we can affirm that the Ray Tracing Problem is also undecidable.

3 Code description

3.1 Main structure

The main structure of our code is split into 3 main files. We will describe the author and content of each of those files and how the code works in its globality.

3.2 sketch.js

This file holds the main code to be run. Its the center of our project and handles everything graphics related. The author for this file is mixed.

3.3 turingMachine.js

This file holds a prototype Turing Machine class we can use to showcase the actions of the Turing Machine that we seek to simulate using rays of lights. This class is heavily inspired from an article by Chad Palmer[2], which describe a simplified and understandable Machine Turing class, which was modified to fit more easily in our code. The main author of this file Emile MERIAN.

3.4 visualizer3D.js

This file holds all the code necessary to model the optical system we wish to present. The file presents all the tools necessary to create the simple boxes of the optical system, where they are placed by the sketch.js file. The main author of this file is Elliot SILBERWASSER.

This file uses the THREE.js lib which allows us to generate a 3D environment in javascript.

THREE.js also allows management of a free camera, called OrbitControls, very useful for our presentation in order to be able to zoom and concentrate on the different boxes. A main HTML container is designed for the site in order to guarantee total control of camera movements.

The way the project was designed is a hybrid between static code and dynamic code. All the box creation functions allow you to create a box at a given x, y, z position. In this way we can recreate the both implementations of the rightMove and leftMove transitions box by box. By static we mean that the ray is not directly reflected by the different planes representing completely reflective surfaces. Indeed, for reasons of time and the desire to propose a project which allows us to really see the impact of the different boxes on the laser, we preferred to favor static. On the other hand, for the dynamic side, we still implemented for the left move, different possible positions of the laser allowing us to visualize the impact of the position of the laser at the input at state q on the output q'. For a U value of 0.75, the possible values of V represented in our app are $\{0.5, 0.25, 0.75, 0.125, 0.875\}$. The arbitrary choice of U was determined in order to propose the leftMove in the same direction as the article in order to maintain a certain consistency.

4 Bibliography

References

[1] J. D. Tygar J. H. Reif and A. Yoshida. "Computability and Complexity of Ray Tracing. (English)". In: *Discrete Comput Geom 11:265-287* (1994). DOI: https://link.springer.com/content/pdf/10.1007/bf02574009.pdf.

[2] Chad Palmer. "A Complete Web Page: Building a Turing Machine in JavaScript". In: (2020). DOI: https://medium.com/swlh/a-complete-web-page-building-a-turing-machine-in-javascript-d6c32d3708c4.