Texture Packing

Group number: Jiefeng Wu Jiajun Qin Wenjie Huang

May 27, 2023

Abstract

Texture packing is a technique used in computer graphics and game development to efficiently organize and store multiple smaller textures or images into a larger texture, often referred to as an atlas or sprite sheet. The goal is to minimize wasted space and improve rendering performance by reducing the number of texture switches during rendering.In order to get an approximate solution to the Texture Packing problem in polynomial time, we use Next Fit and First Fit algorithms, and through experiments and analysis, we know that the optimality of the NF algorithm is not as good as the FF algorithm, but the time complexity of the NF algorithm is much smaller than that of the FF algorithm, in addition, sorting the input data first and then using the FF algorithm will improve the optimality of the results to a certain extent, but when the input is large, Sorting the input data also adds a lot of time. Therefore, our project provides a reference to some extent on how to choose algorithms to deal with texture packing problems

1 Introduction

Texture Packing is to pack multiple rectangle shaped textures into one large texture of a given width and requires the resulting texture to have a minimum height. This problem involves designing an approximation algorithm that runs in polynomial time, generating test cases of different sizes(different widths and heights) and analyzing all the factors that might affect the approximation ratio of our proposed algorithm is expected.

2 Data Structure / Algorithm Specification

2.1 Data Structure

In this project, We use a rectangle class to encapsulate the input rectangle shaped textures and the resulting texture. Besides, we use a vector array to store objects of the rectangle class.

The definition of the rectangle class

```
1     class Rectangle {
2     public:
3         Rectangle(int a, int b): height(a), width(b) {}
4         int GetHeight() {return height;}
5         int GetWidth() {return width;}
6         void SetWidth(int a) {width = a;}
7     private:
8         int height;
9         int width;
10    };
```

2.2 Algorithm Specification

In this project, we use two approximation algorithms to solve this problem, which are Next Fit algorithm [1] and First Fit algorithm [2]. In this two algorithms, we assume that all rectangular blocks(rectangle shaped textures) are stored in rows, which means that there will be no rectangular blocks stacking on top of each other within a row.

Next Fit Algorithm

For each line, we maintain some attributes:

- now_width - the width for this line
- now_height - the total height (except the now line)
- max_height - the maximum height for this line(can be updated during insertion of this line)

So we try to insert the rectangle into this line, if ok we just update some attributes. Otherwise, we close this line and start a new line.

The pseudo-code of our NF algorithm

```
1     function Pack_NextFit():
2         now_width = 0
3         now_height = 0
4         max_height = 0
5
6         for each tmp in data:
7             if tmp.GetWidth() + now_width <= max_width:
8                 now_width = tmp.GetWidth() + now_width
9                 max_height = max(max_height, tmp.GetHeight())
10            else:
11                now_height += max_height
12                max_height = tmp.GetHeight()
13                now_width = tmp.GetWidth()
14
15        now_height += max_height
16
17        return now_height
```

First Fit Algorithm

For each line, we maintain some attributes:

- now_width - the width for this line
- now_height - the total height (except the now line)
- max_height - the maximum height for this line(can be updated during insertion of this line)

We maintain a free list for the line that has been closed.For a new rectangle, we need first to scan the free list. Choose the first one that can contain the rectangle. If no suitable one, we should try to insert it into the now line. If it's not done yet, we must create a new line and insert. Meanwhile, creating a new line means closing the old line, so we should add the old line into the free list.

The pseudo-code of our FF algorithm

```
function Pack_FirstFit():
    define a vector named free_list to store rectangles
    set now_width to 0
    set now_height to 0 and max_height to 0

    for each tmp in data:
        set bj to false

        for each i in free_list:
            if i's height is greater than or equal to tmp's height and i's
            width plus tmp's width is less than or equal to max_width:
                update i's width to i's width plus tmp's width
                set bj to true (found a suitable position)
                break from the loop

        if bj is true, continue to the next iteration

        if tmp's width plus now_width is less than or equal to max_width:
            update now_width to now_width plus tmp's width
            update max_height to the maximum of max_height and tmp's height

        else:
            if now_width is less than max_width:
                add a rectangle with height equal to max_height and width equal
                to now_width to free_list (for a closed line)

                update now_height to now_height plus max_height
                update now_width to tmp's width
                update max_height to tmp's height

    update now_height to now_height plus max_height (the remaining height for
    the newest line)
    return now_height
```

3 Testing Results

We test the run time of our program using different input sizes. (The width and height of the rectangle shaped texture are randomly generated and the input size is between 10 and 10000.)
Without losing the generality, we set the maximum width in all test cases to 100.

Run time table for NF Algorithm in Table 1:

Table 1: NF Algorithm

Sizes(N)	**Number of repetitions(M)**	**Time(ms)**
10	1000	0.001
50	1000	0.001
100	1000	0.001
250	1000	0.003
500	1000	0.006
750	1000	0.011
1000	1000	0.012
2500	100	0.03
5000	100	0.07
7500	100	0.11
10000	100	0.14

For the FF algorithm, we divide into two different cases depending on whether or not the input data is sorted.
Run time table for FF Algorithm(not sorted) in Table 2:

Table 2: FF Algorithm(not sorted)

Sizes(N)	**Number of repetitions(M)**	**Time(ms)**
10	1000	0.001
50	1000	0.008
100	1000	0.014
250	1000	0.103
500	1000	0.290
750	1000	0.656
1000	1000	1.071
2500	100	5.39
5000	100	28.65
7500	100	39.81
10000	100	75.95

Run time table for FF Algorithm(sorted) in Table 3:

Table 3: FF Algorithm(sorted)

Sizes(N)	**Number of repetitions(M)**	**Time(ms)**
10	1000	0.002
50	1000	0.009
100	1000	0.019
250	1000	0.051
500	1000	0.328
750	1000	0.851
1000	1000	1.315
2500	100	3.72
5000	100	34.21
7500	100	70.14
10000	100	135.74

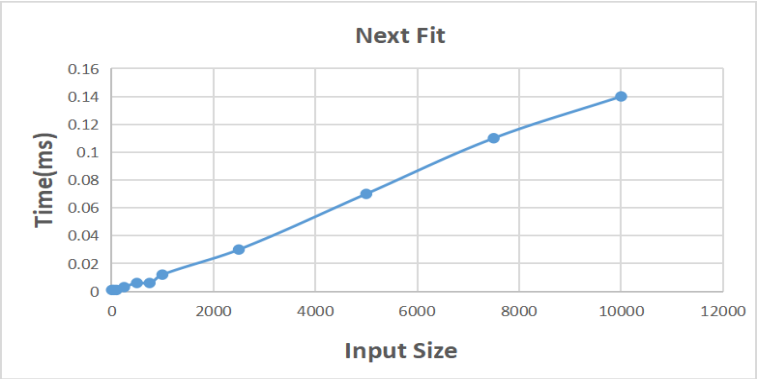


Figure 1: Next Fit Algorithm Run Time Chart

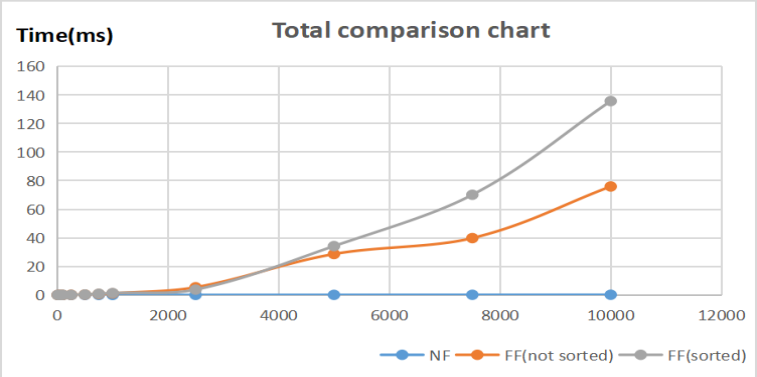


Figure 2: Total Run Time Chart

4 Analysis and Comments

4.1 approximation ratio

To calculate the approximation ratios of the Next Fit and First Fit algorithms for the Texture Packing problem, we need to define a measure for the quality of the packing solution. In this case, we can use the height of the resulting texture as the measure, where a lower height indicates a better packing.

The approximation ratio (AR) of an algorithm A is defined as:

AR(A) = (H(A) - OPT) / OPT

where H(A) is the height of the solution produced by algorithm A, and OPT is the height of an optimal solution.

To calculate the approximation ratios for the Next Fit and First Fit algorithms, we need to compare their resulting heights to the optimal height. However, without knowing the optimal height for each specific instance of the problem, we cannot provide the exact approximation ratios.

Instead, we analyze the factors that might affect the approximation ratios of these algorithms:

1. **Rectangle distribution:** The distribution of widths and heights in the input instances can impact the performance of the algorithms. Some distributions may allow for better packing than others, leading to lower approximation ratios.

2. **Width and height constraints:** The maximum width of the texture and the minimum height requirement

influence the packing results. Tighter constraints may make it more challenging to achieve optimal packing and result in higher approximation ratios.

3. **Algorithm strategy:** Next Fit and First Fit employ different strategies for placing rectangles. Next Fit places each rectangle in the current line until it cannot fit, while First Fit attempts to find space in previous lines before creating a new one. The differences in these strategies can affect the packing efficiency and subsequently the approximation ratios.

4. **Input size:** The size of the input instances can also impact the approximation ratios. Larger instances with more rectangles may be more difficult to pack optimally, resulting in higher approximation ratios.

In general, the **Next Fit algorithm** tends to have higher approximation ratios, especially when dealing with rectangles that have smaller widths. This is because Next Fit places each rectangle in the current line until it cannot fit, which may result in many shorter lines being created, thereby increasing the total height. Therefore, the approximation ratio of Next Fit algorithm can **approach or even exceed 2**. Compared to Next Fit, the **First Fit algorithm** typically has better approximation ratios. By attempting to find suitable space in previous lines first, the First Fit algorithm can better utilize the available space and reduce the number of additional lines. Therefore, the approximation ratio of First Fit algorithm can **be closer to 1**, but it still depends on the specific instance and problem constraints.

4.2 Time Complexity

Next Fit Algorithm

For the NF algorithm, its main time overhead is to determine whether the row can be inserted, this time complexity is O(1), and the time complexity of other operations is also O(1), so in general, the time complexity of the algorithm is O(1).

First Fit Algorithm

For the FF algorithm, its main time overhead is to determine whether it can be inserted into the previous empty row, this time complexity depends on the number of empty rows m (that is, O(m)), and the number of empty rows will change with the increase of N, and due to the influence of chance, the size of its value is difficult to quantitatively express as a function of N, and its time complexity can only be estimated between O(N) and O(N^2) based on $1 <= m <= n$.
Besides, as N increases, for the FF (sorted) algorithm, the proportion of sorting the input data first will also increase, so the time complexity of FF(sorted) should be between O(N$logN$) and O(N^2).

4.3 Space Complexity

For both algorithms, their main spatial overhead is to assign a Rectangle object to the input rectangle element, so the spatial complexity of both is O(N).

5 Author list

The code and report are finished by all of us.

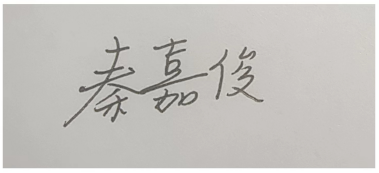
Jiajun Qin finish the code.
WenJie Huang and JieFeng Wu finish the report.

Declaration

We hereby declare that all the work done in this project titled "Texture Packing" is of our independent effort as a group.

6 Signatures

Each author must sign his/her name here.



References

[1] Y. Chen. *ADS11ApproximationStu. page 5.*

[2] Y. Chen. *ADS11ApproximationStu. page 7.*

A Source Code (if required)

main.cpp

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <time.h>
#include "defs.h"
using namespace std;
int n, max_width;
vector<Rectangle>data; /* store the data we generate */
bool cmp (Rectangle A, Rectangle B) /* used for sorting(by height) */
{
    return A.GetHeight() > B.GetHeight();
}
void Print(string msg, int (*func)(), bool sorted)
{
    double st = clock();
    /* For A3, we need to sort first. */
    if (sorted) sort(data.begin(), data.end(), cmp);
    int ans = (*func)();
    double ed = clock(); /* count time */
    cout << msg << " Answer: " << ans << "    " << "Time: " << (ed-st) << endl;
    /* Unit: ms */
}
int main()
{
    cout << "Please enter the number of rectangles N and the width of the box M " <<
    cin >> n >> max_width;
    /* generate ramdom data */
    GenerateData(n);
    /* Algorithm 1: Next Fit */
    Print("NextFit", Pack_NextFit, false);
    /* Algortihm 2: First Fit */
    Print("FirstFit(not sorted)", Pack_FirstFit, false);
    /* Algorithm 3: First Fit with sorting */
    Print("FirstFit(sorted)", Pack_FirstFit, true);
    system("pause");
    return 0;

}
```

defs.h

```
#ifndef _DEFS_H_
#define _DEFS_H_
```

```
3       #define MAX_HEIGHT 100
4       #define MAX_WIDTH 100
5
6       class Rectangle {
7       public:
8           Rectangle(int a, int b): height(a), width(b) {}
9           int GetHeight() {return height;}
10          int GetWidth() {return width;}
11          void SetWidth(int a) {width = a;}
12      private:
13          int height;
14          int width;
15      };
16
17      // data.cpp
18      void GenerateData(int size);
19
20      // next_fit.cpp
21      int Pack_NextFit();
22
23      // first_fit.cpp
24      int Pack_FirstFit();
25
26      #endif
```

data.cpp

```
1       #include <bits/stdc++.h>
2       #include "defs.h"
3       using namespace std;
4       extern vector<Rectangle>data;
5       extern int max_width;
6       void GenerateData(int size)
7       {
8           data.clear();       /* Clear data */
9           srand((unsigned)time(NULL));
10          int i;
11          for (i = 0; i < size; i++) {
12              int a = ((rand() % MAX_HEIGHT) * (rand() % MAX_HEIGHT)) % MAX_HEIGHT;
13              int b = ((rand() % max_width) * (rand() % max_width)) % max_width;
14              if (a == 0 || b == 0) {     /* width/height 0 is invalid data, we skip it
15                  i --;
16                  continue;
17              }
18              data.push_back(Rectangle(a, b));
19          }
20          ofstream out("./data.txt");     /* output data so that we can debug. */
21          out << size << endl;
22          for (auto tmp : data) {
23              out << tmp.GetHeight() << " " << tmp.GetWidth() << endl;
24          }
25      }
```

next_fit.cpp

```cpp
#include <vector>
#include "defs.h"
using namespace std;
extern vector<Rectangle>data;
extern int max_width;
#define max(a,b) ((a>b)?(a):(b))

/*
    Next Fit Strategy.
    For each line, we maintain some attributes.
    now_width - the width for this line.
    now_height - the total height (except the now line)
    max_height - the maximum height for this line(can be updated during insertion of
    So we try to insert the rectangle into this line, if ok we just update some attri
    Otherwise, we close this line and start a new line.
*/
int Pack_NextFit()
{
    int now_width = 0;
    int now_height = 0, max_height = 0;
    for (auto tmp : data) {
        if (tmp.GetWidth() + now_width <= max_width) {  /* We can put it in the now l
            now_width = tmp.GetWidth() + now_width;
            max_height = max(max_height, tmp.GetHeight());
        }
        else {          /* Create a new line.(close the preivous one) */
            now_height += max_height;
            max_height = tmp.GetHeight();
            now_width = tmp.GetWidth();
        }
    }
    now_height += max_height;   /* The rest height for the newest line. */
    return now_height;
}
```

first_fit.cpp

```cpp
#include <vector>
#include "defs.h"
using namespace std;
extern vector<Rectangle>data;
extern int max_width;
#define max(a,b) ((a>b)?(a):(b))

/*
    First Fit Strategy.
    We maintain a free list for the line that has been closed.
    For a new rectangle, we need first to scan the free list. Choose the first one th
    If no suitable one, we should try to insert it into the now line. If it's not don
    and insert. Meanwhile, creating a new line means closing the old line, so we shou
*/
int Pack_FirstFit()
{
    vector<Rectangle>free_list;
```

```
18          int now_width = 0;
19          int now_height = 0, max_height = 0;
20          for (auto tmp : data) {
21              bool bj = false;
22              for (auto i : free_list) {      // Try to find whether we can put it in th
23                  if (i.GetHeight() >= tmp.GetHeight() && i.GetWidth() + tmp.GetWidth()
24                      i.SetWidth(i.GetWidth() + tmp.GetWidth());      /* update */
25                      bj = true;      /* find it */
26                      break;
27                  }
28              }
29              if (bj) continue;
30              if (tmp.GetWidth() + now_width <= max_width) {      /* Put it in the now l
31                  now_width += tmp.GetWidth();
32                  max_height = max(max_height, tmp.GetHeight());
33              }
34              else {      /* now and previous lines cannot used. Create a new line. */
35                  if (now_width < max_width) {
36                      /* For a closed line, we should add it to the free list. */
37                      free_list.push_back(Rectangle(max_height, now_width));
38                  }
39                  now_height += max_height;
40                  now_width = tmp.GetWidth();
41                  max_height = tmp.GetHeight();
42              }
43          }
44          now_height += max_height;       /* The rest height for the newest line. */
45          return now_height;
46      }
```