Horse Carriage - Group 16

Darsh Shah
120010010
darshs@cse.iitb.ac.in

Ayush Deothia
120050025
ayushdeothia@cse.iitb.ac.in

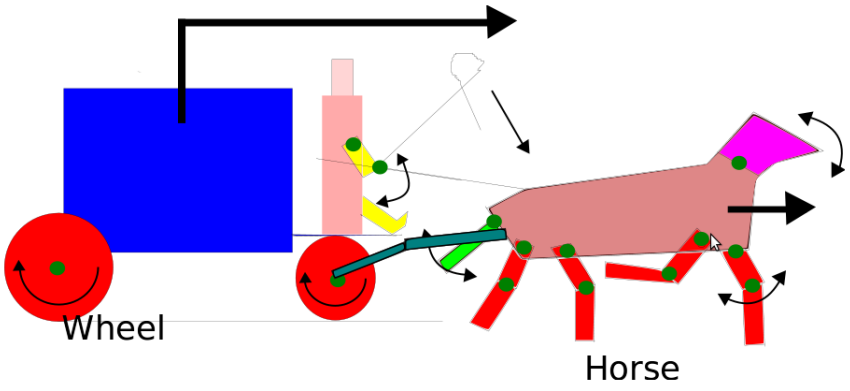
Dheeraj Reddy
120050061
redmond@cse.iitb.ac.in

April 10, 2014

Introduction

This report has been made, to present our project for the cs296 lab.

Original Design



Brief Description

This, horse carriage is our original Project Proposal.It consists of a horse driven cart, and a man steering it. So, it broadly consists of 3 parts
I) The Horse

- movable body with degree of freedom along the X-axis
- 2 wheels, with ability to rotate and to travel linearly forward
- rod connecting the carriage to the horse

II) The Man

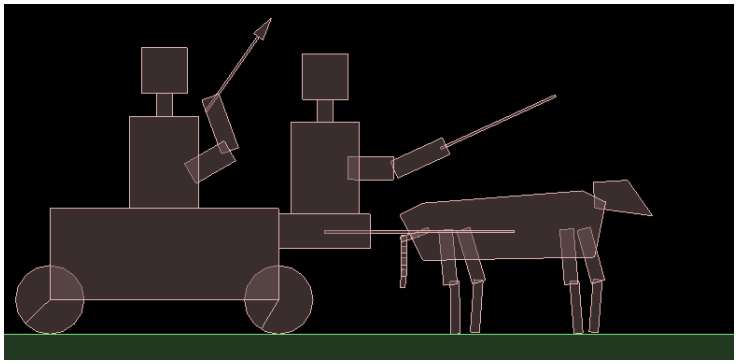
- Movable hand, holding a whip to start the cart by hitting a horse

III) The Horse

- Body, this is the main mass of the horse, and moves along with his legs

- 4 legs, the engine of our transport system, moves ahead and also allows rotation in knees
- Tail of the horse, moves at will
- Horse's head, nods

Final Box2d Simulation



The Final Project

As, can be seen from the pictures above, and the simulation, the project has ticked all the boxes in terms of the initial proposal.We have also had time to make an adition to the original design.
I) ADDITIONS : The Warrior

- Sits on the chariot, and moves ahead
- Holds a spear in his arm
- Releases the spear in a specified direction when given a keypress

II) INTERESTING FEATURES

- The horses movements have been made continuous and are well synchronised, thanks to changes in step function. All stuff about Box2d are referred from [2] and [3].
- The horse's tail is a delight to watch, moves on its own
- Arm movements of both men are very natural

Analysis of Graphs of The Project

Graph 1: Plot depicting Average Step and Loop times (vs) iteration number

The loop-time increases as the iteration number increase, and it is not difficult to see why. For every additional loop iteration, more time is consumed. And therefore as the iterations increase, the loop time also increases.
The average steptime, is seen better in Graph 2.
x iterations of the loop, consists of x distinct steps of our simulation. Thus the average is the sum of times of all x steps divided by x. Since the graph is not a horizontal curve, we can easily conclude that all steps of our simulation are not taking the same amount of time.Initially the average step time is decreasing, implying that time taken per step decreases for the initial steps . But after some 800 iterations, the graph starts to rise, meaning that step 800 onwards, each step of the simulation is taking more time.

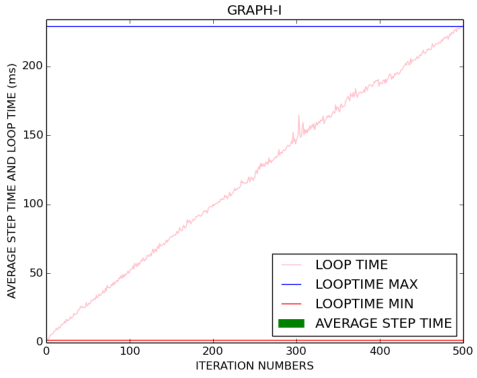


Figure 1: *

Graph 1

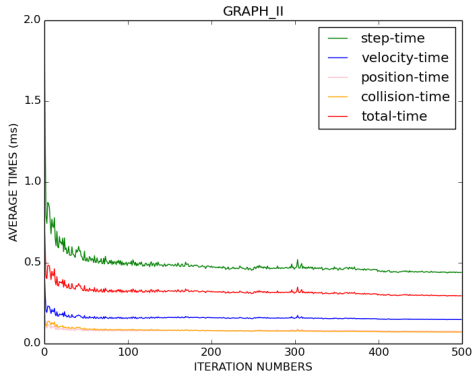


Figure 2: Graph 2

L=Loop time, i=time taken for ith step, N=Iteration Number and S=average step time

$$L = \sum_{i=1}^{N} i \tag{1}$$

$$S = L/N \tag{2}$$

Graph 2: Plot depicting Average times (vs) iteration number

The slopes of these average times i.e Average Step (S), Vellocity (V), Collision (C) and Position time(P) are similar to each other.This shows that averge step time includes these individual times, and its behaviour is very much dictated by these individual components of it.
The sum time(X) is

$$X = P + C + V \tag{3}$$

And the time taken by the Velocity is more than that by the Collision and Position iteration calculations. And this is why the Sum curve and the Velocity curve are adjacent.

Analysing these graphs for a heavy system

We ran the executable with the system being in a heavily loaded (90 %)state. As it was taking significantly more time than it took earlier, we have run only upto 100 iterations. Actually on carefully monitoring the cpu using *top* command we observed that the cs296_16_exe sometime takes a large chunk of cpu in other time the programs share the space that might explain the large irregularities in the graph.

Graph 3: Plot depicting Average Step Error Bars (vs) iteration number

When the iteration number is high the input sent to main.cpp being larger, the output is an average over a greater number. Thus the error nullifies when averaged over larger samples, and hence the error output attained at higher iterations is lower. This explains the decreasing error as the no. of iterations increase.

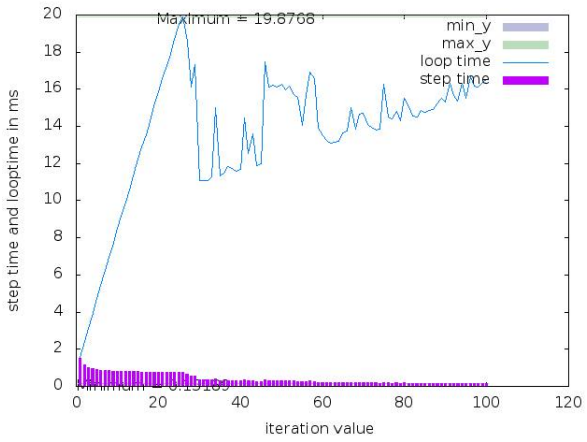


Figure 3: Graph 1 heavy

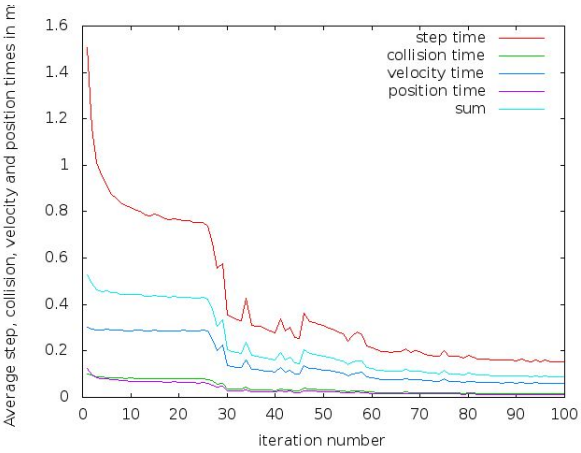
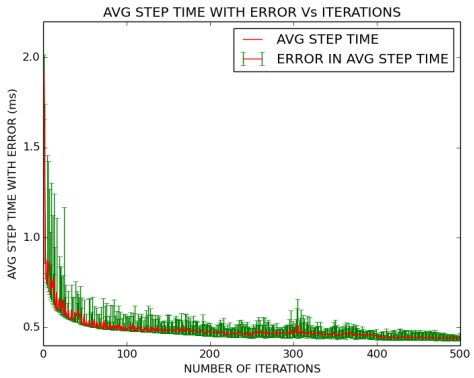


Figure 4: Graph 2 heavy

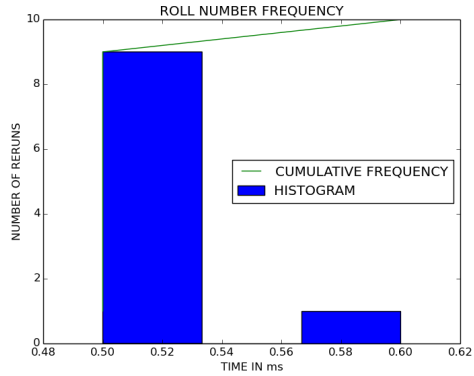
Graph 3



Graph 4: Depicting The Frequency Plot For Avg Step Time

Most of the avg step times(for various rerun numbers) lie in the same bucket as the actual avg step time in our Histogram, whereas the points lying outside this bucket may have occured due to *experimental errors*.

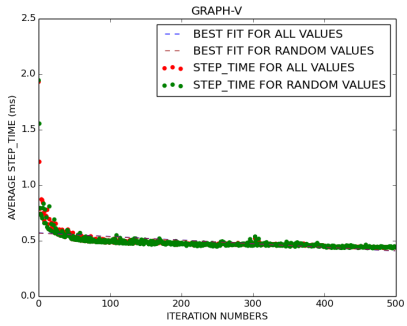
Graph 4



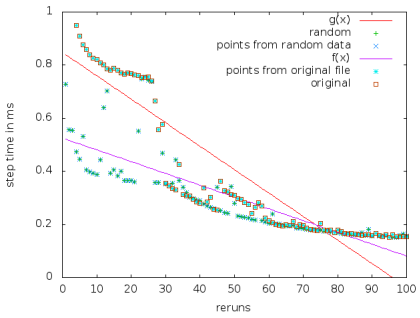
Graph 5: Comparing Average Step Time of Full Sample and Random Sample

When the number of iterations are large, it can be observed that the two graphs converge to a similar function and hence the best fit line is almost the same. This is due to the fact that when the iterations are huge, the number of data points is large and hence the error converges to zero. On the other hand, when we took a lower number of iterations, we could see the difference in the best fit lines of the two.

Graph 5



Graph 5 for smaller data



Understanding the time and gettimeofday()

The time command, gives us three separate times - real,user and sys. The real time gives us the total time passed, when a particular operation is being run along with the time command. The user time, gives us the time required by that particular opertion and does not include the times of other applications that might be running along with the timed operation. In cases where the computer is not loaded with other operations, these two are fairly the same, although for obvious reasons real time is slightly more than user time. sys time gives us the time the operation spends in interacting with the memory and other components of the OS.
The gettimeofday function gives the current time, and hence difference between two gettimeofday functions, at the begining and end of the operation shall work similar to the real time of the time command.

Analysing difference between the two cmake commands Debug Mode and Release Mode

- On running the executable for 100000 iterations, in both the Modes, it is quite easy to see that the loop time required for the Debug Mode (95.92 s) is more than that of the Release Mode (9.38 s) with -O3 flags. We

have observed the .dat files of the two to get more details.

- Compared to the debug data, the release data has skipped quite a few functions. Also, the number of calls made to each function, is far less in the release mode than in the debug mode,as a result of optimization flags. One also sees, that each function takes more time in debug mode than in release mode. For example the function **b2Vec2::b2Vec2(float, float)** takes 0.31 s in the release mode and 2.31s in debug mode.Operator functions also take one-tenth of the time in release mode of what they take in debug mode.

- The operator functions on the b2Vec2 variables also take a humungous amount of time in the debug mode compared to the release mode, where they take no noticable time.

- Several functions that are not even called in the release mode, take around 2s of the time in the debug mode.

- The -On flags also play a very important role.In absence of optimization option, compiler tries to decrease the cost of compilation. When we enable those flags (eg. -O2 -O3), compiler optimizes the performance at a cost of compilation time.

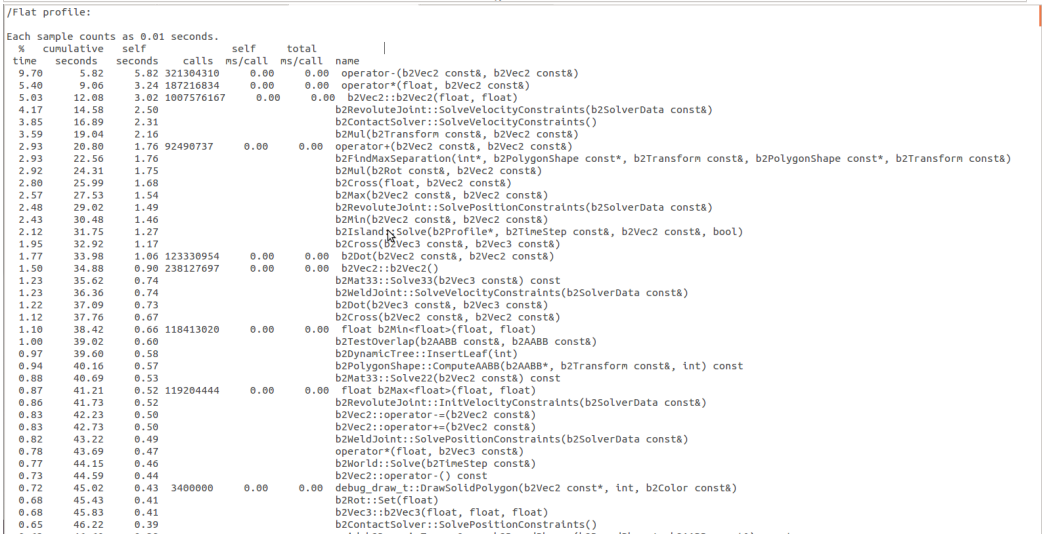
- In release mode, we enable -O3 flags and it tries to do optimization. Optimizations can greatly complicate debugging and hence it takes more time while compiling.But When we run it for 100000 iterations, the cumilative time is only 3.11s. For debug mode, we remove -On flags. Now, the compiler wants to decrease the cost of compilation and to make debugging produce the expected results. It does not optimize the binary it produces. Hence, it takes relatively less time for compiling. But for same number of iterations, cumilative time comes out to be 59.98s (much larger than 6.37s).

- The method, b2RevoluteJoint::SolveVelocityConstraints(b2SolverData const&), takes maximum time in both the modes, 12.72 % (0.81 s)in release mode and 4.17 % in debug mode (2.50 s) . Optimizing this code could help in improving the time drastically.

- In a previous version of the code, we also noticed that since we were running the set angle for a lot of iterations, while making the hand movement, we were utilising more than double the time we are geting currently. Using the step function not only helped in the improvement of graphics of the code, but also made it more optimised. These data have been generated using scripts of previous labs [1]

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 12.72      0.81     0.81                             b2RevoluteJoint::SolveVelocityConstraints(b2SolverData const&)
  8.32      1.34     0.53                             b2ContactSolver::SolveVelocityConstraints()
  7.06      1.79     0.45                             b2RevoluteJoint::SolvePositionConstraints(b2SolverData const&)
  5.34      2.13     0.34                             b2CollideEdgeAndPolygon(b2Manifold*, b2EdgeShape const*, b2Transform const&, b2PolygonShape const*, b2Transform
const&)
  4.87      2.44     0.31                             b2ContactSolver::SolvePositionConstraints()
  4.40      2.72     0.28                             b2DynamicTree::InsertLeaf(int)
  4.40      3.00     0.28                             b2Island::Solve(b2Profile*, b2TimeStep const&, b2Vec2 const&, bool)
  4.40      3.28     0.28                             b2Mat33::Solve22(b2Vec2 const&) const
  4.08      3.54     0.26                             void b2DynamicTree::Query<b2BroadPhase>(b2BroadPhase*, b2AABB const&) const
  3.77      3.78     0.24                             b2Mat33::Solve33(b2Vec3 const&) const
  3.30      3.99     0.21                             b2WeldJoint::SolveVelocityConstraints(b2SolverData const&)
  2.51      4.15     0.16                             b2PolygonShape::ComputeAABB(b2AABB*, b2Transform const&, int) const
  2.20      4.29     0.14                             b2WeldJoint::SolvePositionConstraints(b2SolverData const&)
  2.20      4.43     0.14                             b2World::Solve(b2TimeStep const&)
  2.04      4.56     0.13                             b2RevoluteJoint::InitVelocityConstraints(b2SolverData const&)
  1.73      4.67     0.11                             b2Body::Dump()
  1.57      4.77     0.10                             b2World::SolveTOI(b2TimeStep const&)
  1.41      4.86     0.09                             b2CollidePolygons(b2Manifold*, b2PolygonShape const*, b2Transform const&, b2PolygonShape const*, b2Transform
const&)
  1.41      4.95     0.09                             b2DynamicTree::RemoveLeaf(int)
  1.26      5.03     0.08  3400000    23.53    23.53  debug_draw_t::DrawSolidPolygon(b2Vec2 const*, int, b2Color const&)
  1.26      5.11     0.08                             b2Distance(b2DistanceOutput*, b2SimplexCache*, b2DistanceInput const*)
  1.26      5.19     0.08                             b2ContactManager::AddPair(void*, void*)
  1.26      5.27     0.08                             b2World::DrawShape(b2Fixture*, b2Transform const&, b2Color const&)
  1.10      5.34     0.07  9900000     7.07     7.07  debug_draw_t::DrawSegment(b2Vec2 const&, b2Vec2 const&, b2Color const&)
  1.10      5.41     0.07                             b2WeldJoint::InitVelocityConstraints(b2SolverData const&)
  1.10      5.48     0.07                             b2Fixture::Synchronize(b2BroadPhase*, b2Transform const&, b2Transform const&)
  0.94      5.54     0.06                             void b2BroadPhase::UpdatePairs<b2ContactManager>(b2ContactManager*)
  0.94      5.60     0.06                             b2ContactManager::Collide()
  0.78      5.65     0.05                             b2DynamicTree::Balance(int)
  0.78      5.70     0.05                             b2ContactSolver::InitializeVelocityConstraints()
  0.78      5.75     0.05                             b2World::DrawJoint(b2Joint*)
  0.78      5.80     0.05                             b2RevoluteJoint::GetAnchorA() const
  0.63      5.84     0.04                             b2TimeOfImpact(b2TOIOutput*, b2TOIInput const*)
  0.63      5.88     0.04                             b2BroadPhase::QueryCallback(int)
  0.63      5.92     0.04                             b2DynamicTree::MoveProxy(int, b2AABB const&, b2Vec2 const&)
  0.63      5.96     0.04                             b2Body::SynchronizeFixtures()
  0.63      6.00     0.04                             b2RevoluteJoint::GetAnchorB() const
```



```
/Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
  9.70      5.82     5.82 321304310     0.00     0.00  operator-(b2Vec2 const&, b2Vec2 const&)
  5.40      9.06     3.24 187216834     0.00     0.00  operator*(float, b2Vec2 const&)
  5.03     12.08     3.02 1007576167     0.00     0.00  b2Vec2::b2Vec2(float, float)
  4.17     14.58     2.50                             b2RevoluteJoint::SolveVelocityConstraints(b2SolverData const&)
  3.85     16.89     2.31                             b2ContactSolver::SolveVelocityConstraints()
  3.59     19.04     2.16                             b2Mul(b2Transform const&, b2Vec2 const&)
  2.93     20.80     1.76 92490737     0.00     0.00  operator+(b2Vec2 const&, b2Vec2 const&)
  2.93     22.56     1.76                             b2FindMaxSeparation(int*, b2PolygonShape const*, b2Transform const&, b2PolygonShape const*, b2Transform const&)
  2.92     24.31     1.75                             b2Mul(b2Rot const&, b2Vec2 const&)
  2.80     25.99     1.68                             b2Cross(float, b2Vec2 const&)
  2.57     27.53     1.54                             b2Max(b2Vec2 const&, b2Vec2 const&)
  2.48     29.02     1.49                             b2RevoluteJoint::SolvePositionConstraints(b2SolverData const&)
  2.43     30.48     1.46                             b2Min(b2Vec2 const&, b2Vec2 const&)
  2.12     31.75     1.27                             b2Island::Solve(b2Profile*, b2TimeStep const&, b2Vec2 const&, bool)
  1.95     32.92     1.17                             b2Cross(b2Vec3 const&, b2Vec3 const&)
  1.77     33.98     1.06 123330954     0.00     0.00  b2Dot(b2Vec2 const&, b2Vec2 const&)
  1.50     34.88     0.90 238127697     0.00     0.00  b2Vec2::b2Vec2()
  1.23     35.62     0.74                             b2Mat33::Solve33(b2Vec3 const&) const
  1.23     36.36     0.74                             b2WeldJoint::SolveVelocityConstraints(b2SolverData const&)
  1.22     37.09     0.73                             b2Dot(b2Vec3 const&, b2Vec3 const&)
  1.12     37.76     0.67                             b2Cross(b2Vec2 const&, b2Vec2 const&)
  1.10     38.42     0.66 118413020     0.00     0.00  float b2Min<float>(float, float)
  1.00     39.02     0.60                             b2TestOverlap(b2AABB const&, b2AABB const&)
  0.97     39.60     0.58                             b2DynamicTree::InsertLeaf(int)
  0.94     40.16     0.57                             b2PolygonShape::ComputeAABB(b2AABB*, b2Transform const&, int) const
  0.88     40.69     0.53                             b2Mat33::Solve22(b2Vec2 const&) const
  0.87     41.21     0.52 119204444     0.00     0.00  float b2Max<float>(float, float)
  0.86     41.73     0.52                             b2RevoluteJoint::InitVelocityConstraints(b2SolverData const&)
  0.83     42.23     0.50                             b2Vec2::operator-=(b2Vec2 const&)
  0.83     42.73     0.50                             b2Vec2::operator+=(b2Vec2 const&)
  0.82     43.22     0.49                             b2WeldJoint::SolvePositionConstraints(b2SolverData const&)
  0.78     43.69     0.47                             operator*(float, b2Vec3 const&)
  0.77     44.15     0.46                             b2World::Solve(b2TimeStep const&)
  0.73     44.59     0.44                             b2Vec2::operator-() const
  0.72     45.02     0.43  3400000     0.00     0.00  debug_draw_t::DrawSolidPolygon(b2Vec2 const*, int, b2Color const&)
  0.68     45.43     0.41                             b2Rot::Set(float)
  0.68     45.83     0.41                             b2Vec3::b2Vec3(float, float, float)
  0.65     46.22     0.39                             b2ContactSolver::SolvePositionConstraints()
```

Inference from The Call-Graphs

- The call graphs give an extremely crisp and easy to understand depiction of the code. Like any graph, it consists of nodes N - which is a particular function and edges E between two nodes (N1,N2).

- It is a directed graph. This is because an edge (N1,N2), will show the calling of the function N2 inside of N1. Each Node also stores, along with its function name, the total fraction of the total time consumed by that function in %, and in brackets the self time i.e., the time the functions spends after removing the time spent via calls to other children function. It also stores the number of times that that function has been called.The edges between (N1,N2) stores the amount of time taken by the function N2, when N1 calls it. It also shows the number of times N1 calls N2.

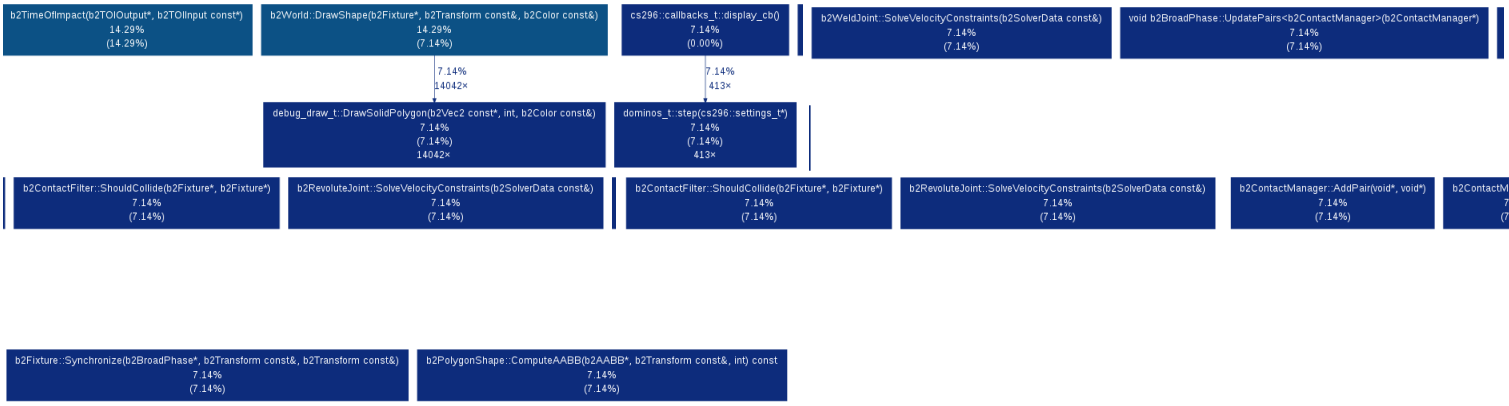
- The call graphs of the two Modes, clearly show why there is such a massive difference in times, when both are run. The debug mode has a lot more functions that are called, and call other functions. In case of release mode,this is relatively less.Also, a lot of functions are not even called here. The self times of the functions, are almost the same as the function times in the release mode. But in the debug mode, the self time is

significantly less the total time.This shows the difference in the optimization and hence the time difference that results.

- Other important use of callgraphs is the color of a node.The color of node gives us the proportionality of time directly so if we want to optimize some function just cheking the legend could be of great help

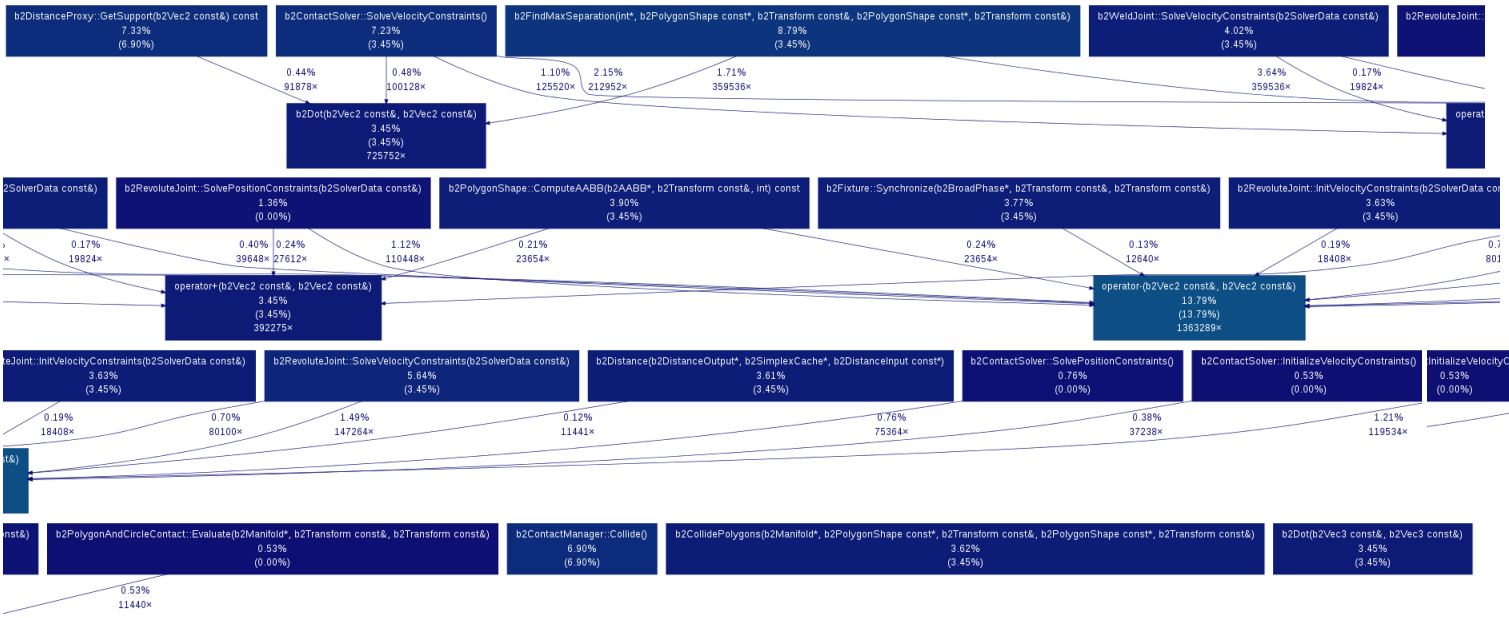
Release Mode

Release mode enlarged



Debug Mode

Debug mode enlarged



References

[1] Python Scripts from previous labs.

[2] Box2d Manual. `http://www.box2d.org/manual.html`.

[3] Box2d C++ tutorial. `http://www.iforce2d.net/b2dtut/`.