IMPROVING THE 2.5D STAGE ROBOTIC SIMULATOR

Nick Wong and *Jui-Chun Peng Hsu* and *Toby H. J. Collett* and *Bruce A. MacDonald*

Department of Electrical and Computer Engineering
University of Auckland, New Zealand

Abstract

The open source Stage simulator is a plug-in for the Player robotic software system, that enables the user to test robotic devices in a simulated environment. While designing and building the hardware for a new robotic system might consume numerous months, the Stage simulator provides for building a simulation world with different robotic devices, and for the execution of different tasks, in a time frame of less than a week.

Recent additions to Stage include a 2.5D version for simulating objects with simple height properties. However, the 2.5D version of Stage is incomplete. This project aims to provide a complete working solution for 2.5D.

Several changes to Stage 2.5D are reported, including improvements to the collision detection and the addition of a gripper device that moves in the z direction. The solution for the collision detection issue provides full 2.5D collision detection functionality and the gripper model is used to allow the robot to complete simple 2.5D tasks such as stacking an object on top of another.

1. Introduction

The Player/Stage project was originally developed by Brian Gerkey, Kasper Stoy, Richard Vaughan and Andrew Howard. The projects were first named Golem and Arena with the intention of improving the woeful state of robot software infrastructure by using UNIX-like abstractions and machinery (Gerkey *et al*, 2003). The project was made public as open-source on Sourceforge in 2001. Since then, it has experienced wide spread use in both academia and industry. Many robots and devices are modeled in Player/Stage, so that together there are approximately 2500 files and 190,000 lines of source code, as reported by CLOC (2008). Player is a networked device server and it is one of the most widely used robot control interfaces in the world. Stage is a simulator originally designed to simulate large populations of mobile robots in 2D simulation worlds.

The development of Stage has been driven by the increasing use of the Player server. Stage is capable of simulating multiple robot interactions in an indoor environment. Stage 2.5D has been released recently and is capable of simulating the interaction between robots in a 2.5D environment. However, the 2.5D functionality of the existing Stage simulator is incomplete. For example, the collision detection assumes all objects are infinitely tall, so that a short robot is unable to go underneath a higher obstacle. The 2.5D Stage simulator project involves developing the physical model of Stage 2.5D so that it is able to provide complete 2.5D functionality.

In a 2.5D environment all the 2D objects have two properties on the Z-axis, a starting location and a size. So objects are effectively 2D shapes extruded along the Z-axis.

The first section describes the scope and specifications and how these were refined throughout the lifespan of the project. The second section mentions related work. The third section describes the project goals. The fourth section explains the methodologies used to carry out the project. The fifth section gives the project specification. The sixth section gives a detailed description of the architecture of Stage. The last section is a detailed explanation of the improvements made to the collision detection and the addition of a gripper model.

2. Related work

Gazebo is the 3D simulator in the Player/Stage project. It is designed to accurately reproduce the dynamic environments a robot may encounter. All simulated objects have mass, velocity, friction, and numerous other attributes that allow them to behave with physical realism (Koenig & Howard, 2004). The complexity of producing such simulation demands substantial computing power and limits Gazebo to simulating only one or two robots. Stage provides a faster and more efficient solution for a simulation world, enabling multiple robot interactions in an indoor environment.

3. Project Goals

The Player project seeks to develop a working Stage plug-in for Player which should provide full 2.5D simulation support. The original Stage version 3 under the source code repository already supports most of the 2.5D rendering functionalities but it is in an alpha state and requires debugging and testing. Our goal was to

find bugs, fix the general limitations for Stage 2.5D, and also provide tests for the 2.5D concept.

3.1. Included Scope

- Report and fix bugs encountered while using Stage version 3.
- Test and evaluate existing software to identify missing features and bugs.
- Contribute to the open source community by providing update patches and report any bugs encountered.
- Enhance the overall 2.5D concept by providing test cases which shows the fundamental interactions between the robot and the simulation world, these are:
  - Objects should be able to be constructed with different heights.
  - Collision detection should be supported for objects with different heights.
  - The robot should be able to drive underneath an object.
  - The robot should be able to perform tasks such as stacking of objects.

3.2. Excluded Scope

2.5D simulation should be the boundary of the application domain to be supported by the new version of Stage. If a user requires full 3D simulation, then other software such as Gazebo should be used instead. Stage should maintain its simplicity and provide users computationally cheap models and the simulation support for a wide variety of devices.

Modifications made to the Player software should be minimal since it is not directly related to the 2.5D models. However, changes to Player may be necessary to incorporate new virtual devices or drivers while testing.

4. Project specification

The project goal is to improve the existing Stage simulator to be capable of simulating a 2.5D environment, and allow the robot to perform 2.5D tasks such as driving under objects and stacking objects on top of each other. The project involves working closely with the Player/Stage open source group and to make contributions to the Player/Stage project.

The project specification was revised regularly as the project progressed. During the evaluation phase of the project, the team decided to focus on the collision detection issue because a significant amount of time was consumed during an initial evaluation of the code. However, resolving the collision detection issue only took half the time allocated for the implementation phase. So after discussing the available options, the team decided to tackle the object stacking task by adding a gripper device model to the existing Stage system.

5. Methodology

The project was conducted by authors Peng Hsu and Wong using an improved version of the waterfall model in three phases: evaluation, implementation and testing. When a phase is fully completed, the team proceeded to the next phase while revising the previous phase regularly. The Player/Stage project is an open source project and is improved frequently. Our approach allowed the project team to quickly adapt to the changes made by other contributors and make appropriate adjustments.

The project files are stored in a subversion (SVN) repository, which provides a secure way to modify and update the project, as well as a version history.

5.1. The evaluation phase

The Stage simulator consists of 588 source files with a total size of 3.5MB. An evaluation phase enabled the project team to gain a basic understanding of the source code, and the functionality of Stage.

During the evaluation phase, Peng Hsu and Wong worked in parallel, studying different parts of the source code. The three areas of source code evaluated were the treatment of the configuration files by lexical analysis, the construction of the simulation world by scanning pixels in a world image file, and collision detection by the voxel traversal ray tracing technique (Heckbert, 2004). The significant findings and related information observed by the members were discussed frequently during and after each code evaluation session. This process brought the two team members a good understanding of the code base.

5.2. The implementation phase

The collision detection function and a gripper model were designed and added to the Stage simulator in the implementation phase. The Player/Stage open source group was consulted before the project team tackled each issue, to coordinate the work with other contributions, and to seek advice.

5.3. The testing phase

The improved Stage simulator was tested by creating several Stage simulation worlds and performing a set of manual tests on them. The test clients are created to perform a certain task and the real behaviors and the intended behaviors are compared and analyzed.

6. Stage architecture

The Stage architecture is graphically depicted in Figure 1. The world represents the set of all models and the

floor plan is constructed by mapping pixels from a bitmap file that represents the static environment. The client sends and receives data from the model interfaces. A model can have a list of children which are also models. A model is represented by a list of rectangle blocks rendered by OpenGL.

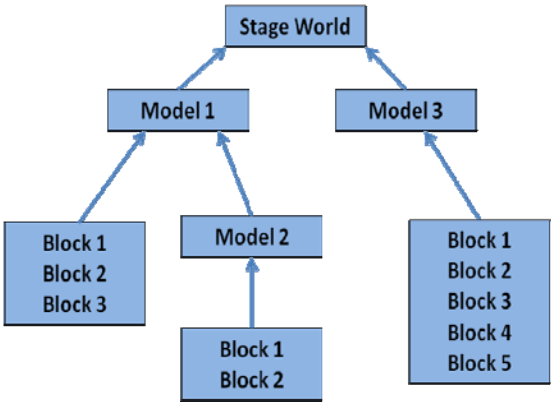

Figure 1: The Stage structure.

6.1. The configuration files

To understand the configuration file Stage uses. We first have to understand the structure of the devices used in Playerstage.

There are three key concepts for each device in Player, interface, driver and device.

The interface specifies the syntax of all messages that can be exchanged in a device class. The driver is a hardware controller which knows how to control a real device on the robot; it receives the command sent by the client and executes the command on the hardware device. Requested data and feedback from the device are then returned to the driver. The driver translates the retrieved raw data from a hardware device to the format specified by the interface and hides the specifics of the given device. The device is the combination of a driver and an interface. A device is created in the configuration file by specifying the driver, the interface and an address. The client can then connect to the proxy of the device at the address and send a message to the driver through the interface of the device.

Figure 2 illustrates the structure of a device using a real laser sensor called E23 as an example.

The configuration file is essentially a list of drivers bound to their interfaces. Different devices are bound to a robot by attaching them on the ‘stage’ driver to the robot model which is defined in the world file. When using Player as a Stage client, the configuration file will contain a 'simulation' interface entry that uses the 'stage' driver and specifies a 'world file' with a '.world' suffix that describes the contents of the world that Stage should simulate.

The ‘stage’ driver also has the libstageplugin library attached to it. The plugin is a shared library which will be loaded before instantiating the ‘stage’ driver. The libstageplugin library contains the device interfaces in Stage which subscribe to the messages from Player’s device interface and process the messages in the Stage simulation.

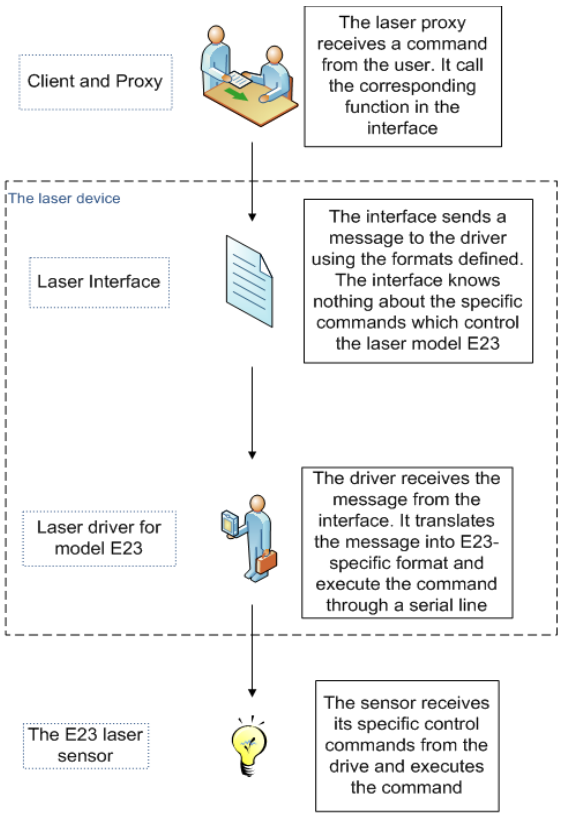

Figure 2: The device structure.

6.2. The world

The world file loads the environment bitmap by mapping the entire file to pixels and grouping the filled pixels into a list of rectangle blocks. The blocks are rendered by Stage’s OpenGL library. The world file also contains the definition of the models in the world. Models must be declared in the world file with parameters such as positions, bearings and colors.

6.3. The model

The graphical presentation of a model is composed of a list of blocks specified in the .inc file. Each .inc file contains the specification for a number of models including the models’ properties and shapes.

To construct a model, one has to first define a number of polygons to approximating the shape of the model. Each polygon’s vertices and its location on the z

axis must be specified and stored in a "blocks" array. The blocks will be rendered in runtime to approximate the model's shape.

A model may contain other models. For example, a pioneer robot model consists of two different component models, the position model which represents the base of the robot and the laser model which represents the laser sensor's appearance and properties. The list of children of a model is stored by the pioneer model as one of its class attributes.

There are 2 common types of model, devices and objects. The .inc file for a device model contains several properties of the device. For example, a laser sensor device contains properties such as the field of view, range and number of samples. A device model also has a parent model. An object model is an object placed in the simulation world. Its properties define how it reacts to other models. For example, an object model with a property "obstacle_return 1" can cause a collision when another model comes in contact with it, and an object model with "laser_return 1" is detectable by a laser sensor.

A model class in Stage contains 2 main functions required by the simulation and a set of other functions used by the model itself. The Load function is called when the model is added to the simulation world, the properties of the model are retrieved from the .inc file and stored in the model class. The Update function is called every time the simulator updates itself. Each model also has its own set of functions. For example, the position model is able to set the speed of the robot and the laser model is able to query the range data.

6.4. The sensor

The sensors are a special kind of model. In a real environment, sensors gather information from the environment for the controller to analyze for useful data. However, in a simulated environment, sensor behavior must be simulated. In the existing Stage, the sensors use a ray tracing technique to simulate sensor behaviors. The ray tracing function uses the height of the sensor to determine the 2D plane in which to look for collisions.

6.5. The simulation

Each robot listens to a specific port defined in the configuration file. Figure 3 illustrates the operation sequence after a command is sent from the client. The client sends a command to a port currently being listened to by the Stage simulator. The command is received by the Player server and then processed by the device interfaces. The interface then calls the corresponding function in the model class and the function is handled by the model class accordingly. The world update function is called periodically in a fixed interval defined in the world file. This function calls all the models' update functions and updates the world simulation. A response is sent from the world if it is required. The response message is returned to the client by the interface publish function.

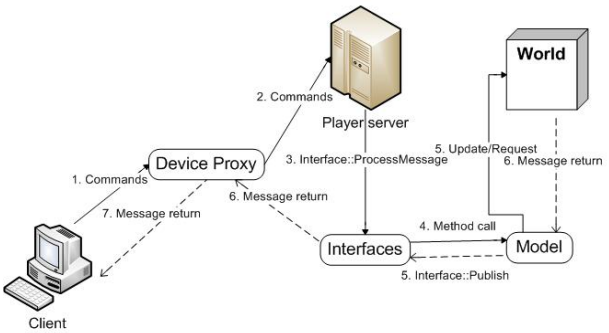

Figure 3: The Stage simulation sequence.

7. Stage improvements

7.1. The existing collision detection

Stage's collision detection is used to send a signal to the client when a model in the simulation collides with another model in the world. Gazebo is a 3D simulator and is capable of simulating complicated environment interactions such as mass, friction and numerous other attributes using the Open Dynamics Engine, so collisions can be played out in the simulation to see what the physical results are. In contrast the Stage project focuses on multiple robot interactions in a simple planar indoor environment. The complexity of simulating realistic behaviors when one object collides with another object can severely tax even a high performance computer. Therefore, when a collision occurs in Stage, the signal simply stalls the simulation. Stage is not expected to simulate the physical results of the collision.

Collision detection uses ray tracing to determine whether a line has intersected with another object in a given range. However, the z range check used by the existing ray tracing function is designed for *sensors*. The collision detection function takes the velocity of a moving model and computes its next position after the next simulation update cycle. The range is then calculated by comparing the next position with the current position. If the ray tracing function returns a hit within the given range, a collision signal is sent and displayed in the simulation world.

Figure 4 illustrates the responsibility of each component involved in the collision detection process.

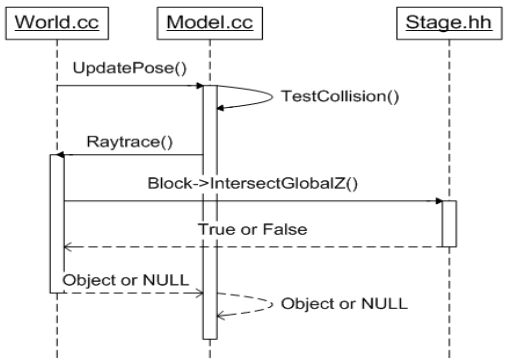

Figure 4: The collision detection sequence.

The simulation calls the world update function repeatedly, which also calls the update pose function on all the models that have a velocity. Before updating the position of the model, the update pose function calls the test collision function which uses the same ray tracing function used by the sensors to check for collision. The test collision function retrieves all the edge poses of the current model's blocks and sets the z value of all the edge poses to 0. The edge poses are then passed to the ray tracing function to test for collision, from a 2D perspective. Hence, it only checks for whether the x, y values of each edge pose intersect with another object. The result from ray tracing is combined with a simple z range check to make sure z is between global maximum and minimum values. Here, z is the height of the edge pose which is by default to 0 and the global minimum and maximum are the z values of a block in the simulation world. The ray tracing function loops through all the models' blocks and tests for collisions. Since the z values for the edge poses are all 0, the check in returns false whenever the block is above the ground level (z > 0) regardless of the height of the block. Hence, the collision detection only works for objects that have a presence at ground level. Figure 5 shows how the existing collision detection fails to signal a collision when the robot collides with a table.

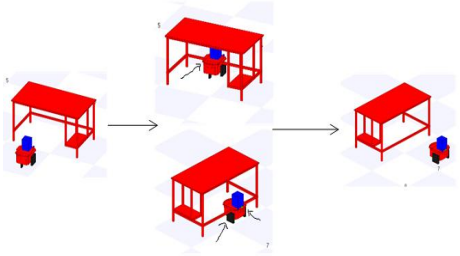

Figure 5: The existing collision detection.

7.2. Improving the 2.5D collision detection

7.2.1. The z range check

Two approaches were considered for solving the z range check problem. The first approach is to modify the ray tracing function and use a fully functional 3D algorithm instead of the existing 2D ray tracing algorithm. Since all the other sensors use the ray tracing function to gather information, the new algorithm would have to be modified to match the structure of Stage, which involves considerable work. Another approach is to add proper checking on the z axis without modifying the ray tracing function. The second approach was suggested to and approved by the open source group. To check the z intersection of two blocks, the z minima and z maxima of both blocks must be determined. Figure 6 illustrates different z intersection scenarios.

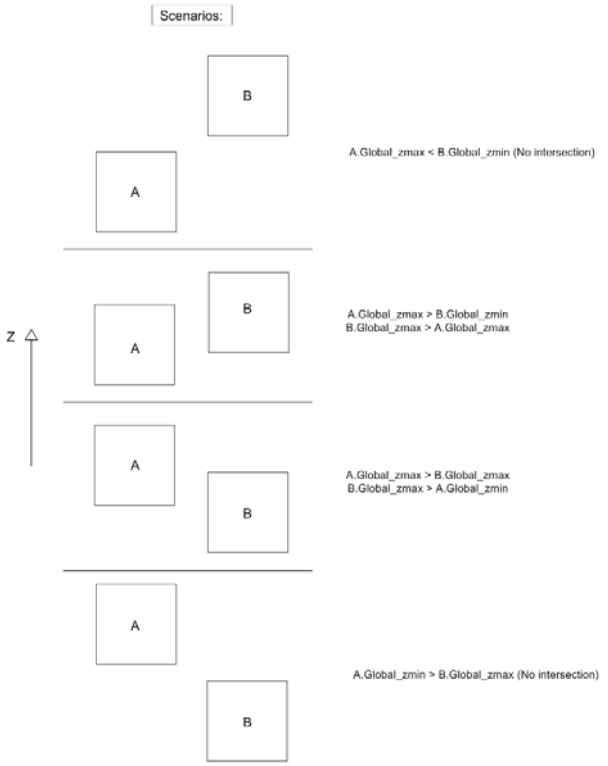

Figure 6: The z range intersections.

The C++ functions for checking z range intersection are overloaded by adding a definition that takes 2 arguments, z minimum and z maximum of a block.

7.2.2. Ray tracing function modifications

To retrieve the z minimum and z maximum values of the block which is being tested for collision, two functions are added to the block class. In a 2.5D environment, the position on the z axis is the same for all the edges of a block. The existing ray tracing function only contains information about the global x, y

pose of the block but not the block itself. Therefore, an extra argument containing a pointer to the block is added to the ray tracing function.

```
void StgWorld::Raytrace(
        stg_pose_t pose, // global pose
        stg_meters_t range, stg_block_match_func_t func, StgModel* mod,
        const void* arg, stg_raytrace_sample_t* sample, bool ztest,
        StgBlock* testBlock) {

    //Check if the raytrace is testing for collision,
    bool collisionTest = false;
    stg_meters_t z_min = 0;
    stg_meters_t z_max = 0;
    // If testBlock != NULL, we are testing for collision,
    if (testBlock != NULL) {
        collisionTest = true;
        ztest = false;
        // Set the z_min and z_max to the block's z values
        z_min = testBlock->Get_global_zmin();
        z_max = testBlock->Get_global_zmax();
    }
```

Figure 7: The z values retrieval.

The extra argument is only used when the ray tracing function is called by the test collision function in model.cc. The argument testBlock is set to NULL when the ray tracing function is used by sensors. As shown by Figure 7, if the testBlock is not NULL, the collisionTest variable is set to true indicating the Raytrace function is testing for collisions. The z values are then retrieved from the testBlock and passed to the z intersection checking function.

7.2.3. Improving the test collision function

It is important to check whether the collision detection function checks the children of each model as well as the model itself.

When the simulation world updates itself, it calls the update pose function on all the models in the velocity list array. The list is created in the world construction function and a models is added to the list if it has a non-zero velocity. Since the pioneer model is added to the world as a whole, its children will not be added to the velocity list and hence they will not be tested by the test collision function.

The first approach to resolve this issue is to group all the blocks associated with a model first, and then use the test collision function to test the blocks for collision. A recursive function is added to the model.cc to group all the blocks associated with a model into a list. The idea is to recursively pass a pointer of a list to gather all the blocks. However, this approach has proven to be problematic. Firstly, appending to list is inefficient and not scalable because it requires traversing the entire list. More importantly, the logic is incorrect; for every recursion, adding a block to the list also adds it to the model itself, which is wrong.

A more efficient approach to solving the problem is for the new test model collision function to use the test collision function by iterating over all the child models associated with the current model, calling the test collision function on each child model. It first checks whether or not the last recursion's test collision method has returned a hit; if true the recursion stops and returns.

Since the test collision function checks the model's blocks against the set of *all* the objects in the simulation world, the current model itself and its children must be un mapped from the simulation world before calling the test collision method, to avoid self hits. The models are then added back to the simulation world.

Initial tests of the improved collision detection function have been successful and the collision detection patch has been submitted to the Player/Stage code base.

Figure 8 illustrates the function call sequence for the improved collision detection.

7.3. Adding a gripper model

7.3.1. Gripper requirements analysis

A gripper is needed if the project goal for stacking objects is to be met. A gripper is defined as a device that is capable of closing around and carrying an object of suitable size and shape. For tasks such as stacking of objects, the gripper must also be able to move vertically to put the objects at different heights. A gripper is typically mounted near the floor on the front of the robot. A gripper typically has two "fingers" that close around an object and is able to carry the object to a storage system or stack objects on top of each other.

The existing gripper device in Player/Stage has been commented out for over two years. While significant changes have been made the gripper code has not been updated. The gripper model is written in C and is not compatible with the current Stage structure. To fix the old gripper model, the project team could have evaluated the old Stage system to find out the intention of some of the functions used in the gripper model. This is a time consuming task since that code base is old. For this reason, the project team decided to rebuild the Stage gripper model from scratch.

7.3.2. Player modifications

As shown in section 6.1, a device consists of a driver and an interface. To add a gripper device to Stage, the driver and interface of the device must be properly defined in Player before attempting to create a gripper model in Stage.

Figure 2 illustrates the progression of a message from the client. The message sent by the client is first received by the Gripper proxy in Player's client libraries. However, the Gripper proxy is not complete. As discussed in the requirement analysis section, the gripper has to be able to move vertically to achieve the tasks such as stacking an object on top of another. Therefore, a "move to height" function must be added to the Gripper proxy to elevate the gripper.

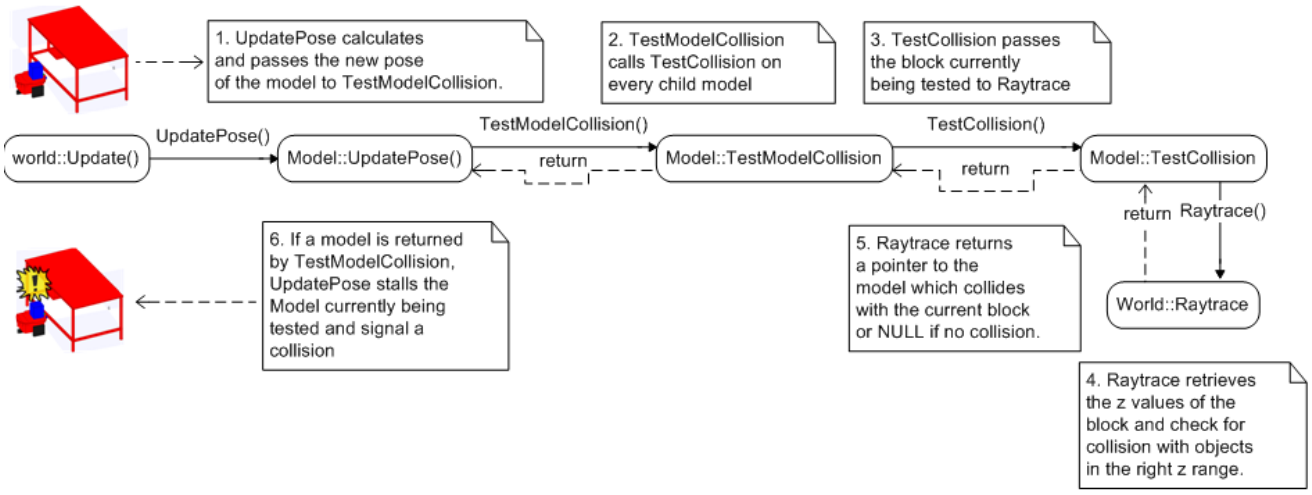

Figure 8: The improved collision detection sequence

The move to height function is responsible for sending the adjust height command to the device, so an adjust height command has to be constructed in the device first.

```
// Command the gripper to change its height
int playerc_gripper_move_to_height_cmd(playerc_gripper_t *device, double percent)
{
    // Not null here, we need to pass a percent value
  player_percent_t cmd;
  memset (&cmd, 0, sizeof (cmd));
  cmd.percent = percent;
  return playerc_client_write
  (device->info.client, &device->info,
  PLAYER_GRIPPER_CMD_MOVE_TO_HEIGHT, &cmd, NULL);
}
```

Figure 9: The new gripper command.

Figure 9 shows the new move to height command in the gripper device. It takes a percent from the client and sends the message to the driver using the format specified by the interface. As we can see from the function, a structure player_percent_t is declared to match the correct message format.

7.3.3. The Stage gripper interface

Each interface in Stage has two functions. Publish and ProcessMessage. The Publish function is used when some data from the device is requested. For example, the client may request a scan using the laser device, the data gathered from the simulation world will then be returned to the client by calling Publish. In order to communicate with the device in Player, Publish uses the same message format defined in the Player's device interface.

According to the requirements, a gripper device does not need to return any feedback about its states. Therefore, the Publish function in the gripper interface sends an empty message back to the client.

The ProcessMessage function subscribes the messages sent by the Player's device interface and processes the message by calling the appropriate functions in the device's model. According to the new gripper device in Player, there are three commands available for the gripper model, gripper_open, gripper_close and gripper_move_to_height. Each message is checked by the MatchMessage function and the matching function in the gripper model is called.

7.3.4. Creating the gripper model class

A gripper model class is created to control the physical model of the gripper in the simulation world. The gripper model class contains 5 functions, Load, Update, GripperOpen, GripperClose and GripperMoveToHeight. Figure 10 illustrates the physical model of the gripper.

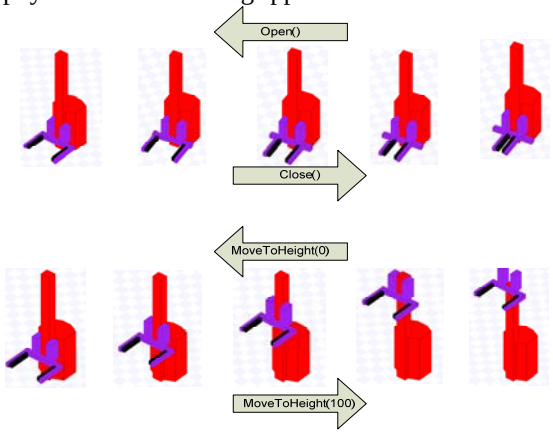

Figure 10: The gripper model.

The idea is to create a set of states for the gripper and every time the Update function is called, it updates the gripper model according to the state it is in.

To keep consistent data structures, the states of the gripper are defined as an enumerated type along with other states used in Stage. Figure 11 shows the states defined for the gripper.

The rest of this section explains how the functions are defined. The GripperOpen, GripperClose and GripperMoveToHeight function change the state of the gripper to opening, closing and moving respectively and the Update function handles the actual operations.

```
typedef enum {
    STG_GRIPPER_LIFT_MOVING = 0,
    STG_GRIPPER_LIFT_STOPPED,
    STG_GRIPPER_CLOSING,
    STG_GRIPPER_OPENING,
} stg_gripper_lift_state_t;
```

Figure 11: The gripper states.

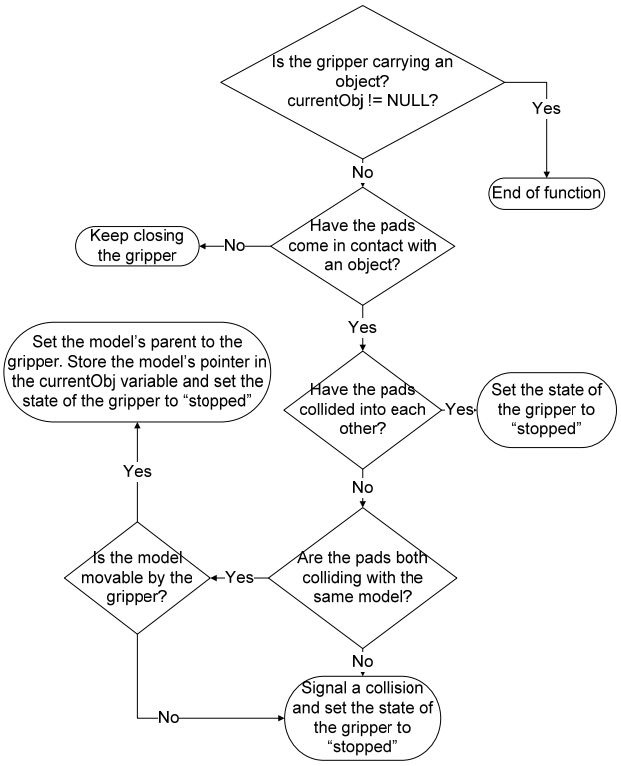

Figure 12: The GripperClose function.

When the gripper is in state stg_gripper_closing, the Update function attempts to close the fingers of the gripper by moving them towards each other.

As shown in Figure 12, the fingers will keep on closing until they collide with an object. Since the physical model of a gripper consists only of blocks, the only way to close a gripper is to move the individual blocks on the gripper model to simulate the closing motion. Therefore, different blocks from the gripper model are retrieved by the Load function and used to animate the closing motion of the gripper. This is an inefficient and error-prone method of closing the gripper and it requires the user to create the blocks in a particular order when building the physical model of a gripper. But the project team could not find a better solution for this problem due to the simple model structure of Stage. The order in which the blocks have to be created is explained in the next section of this report.

To determine whether an individual block has come in contact with another object in the simulation, another collision detection function called TestBlockCollision is defined. The function takes a pointer to a block and returns the block of which it collides with. The TestBlockCollision function is modified based on the TestCollision function. Instead of looping through the model's blocks, it takes a block as input and checks it for collision. The function returns a block if it detects a collision, the block is then checked to see whether it is movable by calling the GetGripperReturn function. A movable model is a model with a property defined as "gripper_return 1" in its .inc file.

When the gripper is in the stg_gripper_closing state, a counter counts the number of times the positions of the gripper's fingers are moved by the Update function and this is used by the GripperOpen function.

After an object is gripped by the gripper, it is time to elevate the gripper to the desired height. Before changing the state of the gripper, the desired height must be calculated. The GripperMoveToHeight function takes a percent and computes the target height specified by the client. The new desired height is calculated by,

$$nH = \frac{p}{100}(mH - iH) + iH \qquad (1)$$

Where the p is the target vertical position in percentage, nH is the new desired height, mH is the maximum height and iH is the minimum height. The maximum and minimum height of the gripper is defined in the gripper's .inc file and are retrieved and stored by the Load function at instantiation.

The vertical speed of the gripper is adjusted by a variable vDistanceDivision that specifies how far on the z axis the gripper moves during a single update call.

After retrieving the information from the gripper mode, the height is updated in the update function.

Figure 13 shows when the gripper is in the stg_gripper_moving state, the update function adjusts the height of the gripper according to the current height, desired height and the moving speed of the gripper.

The GripperOpen function needs a way to determine whether the fingers have reached the initial position. Recall that there is a counter created when the gripper is closing to record the distance by which the fingers

moved. When the gripper is in the stg_gripper_opening state, the Update function decrements the counter and moves the fingers away from each other until the counter reaches 0, so that it accurately positions the fingers back to their original position.

7.3.5. Building the physical gripper model

To build the physical model of a gripper in an .inc file, we first have to specify some properties of the gripper including the size, the maximum height and the vDistanceDivision unit. The maximum height of the gripper is relative to the local position of the robot's base. The vDistanceDivision is the number of updates it takes for a gripper at its initial height to reach its maximum height. A greater vDistanceDivision value makes the gripper to move slower because it takes more updates to reach the given height.

As discussed before, the order at which the blocks are added to the gripper model is crucial to ensure the functions of the gripper are working properly.

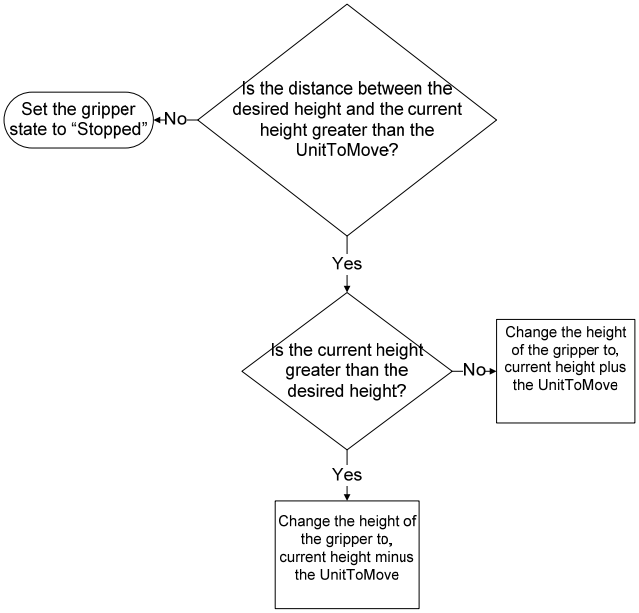

Figure 13: The GripperMoveToHeight function.

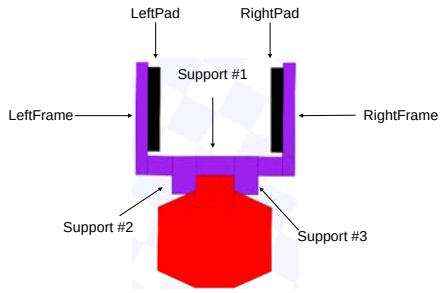

Figure 14: The physical gripper model and its blocks

As shown in Figure 14, a gripper model has 2 frames, 2 pads and a number of support blocks. For a gripper model with n+4 blocks, the frames and pads of the gripper are stored in the last 4 entries on the list. Every time a gripper is loaded, the Load function retrieves the (n+4), (n+3), (n+2) and (n+1) element on the blocks list and assigns the pointers of these entries to RightFrame, LeftFrame, RightPad and LeftPad respectively.

8. Testing

All the tests are done manually by the project team. Several world files have been created for testing.

8.1. The UoA robotics lab

The UoA robotics lab world model is created by measuring the real dimension of the robotics lab at the University of Auckland. The robotics lab world is used to test the collision detection and the gripper device.

A set of obstacle models is created in the world to test the simulator's collision detection function, and a small box model is used to test the new gripper device. In the UoA robotics lab model, robots are moved to collide with other objects in the lab. Objects with different shapes and heights are used to ensure every part of the robot is able to signal a collision appropriately. Multiple robots are also included in the lab to test robots colliding with each other. Figure 15 shows collisions at different heights.

A Player client is created to test the gripper's functions. The client commands the gripper to close on the box and attempts to carry it to a destination by going underneath some objects. After the robot reaches its destination, it lowers itself and drops the box before moving away. The robot's track is shown in Figure 16. The client test checks the 3 main functions used in the gripper device as well as testing the collision detection to ensure it does not signal a collision while going under objects.

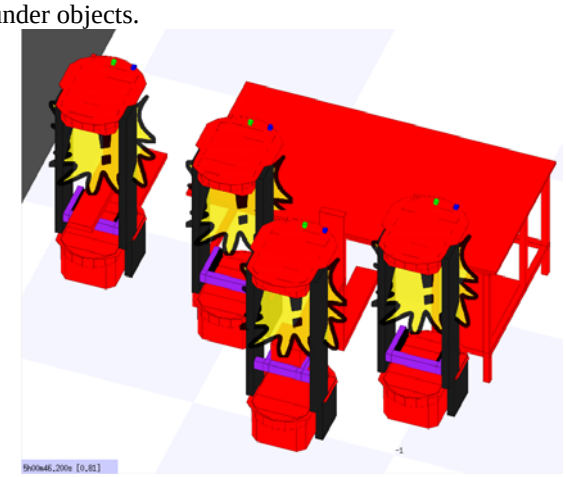

Figure 15: Collision at different heights

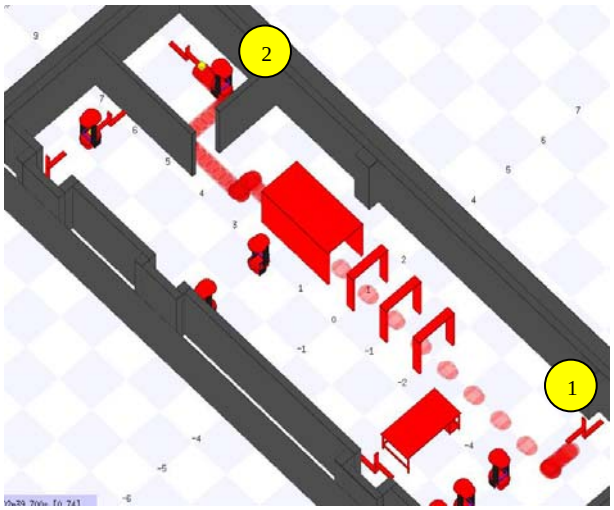

Figure 16: Transporting object and going under tunnels

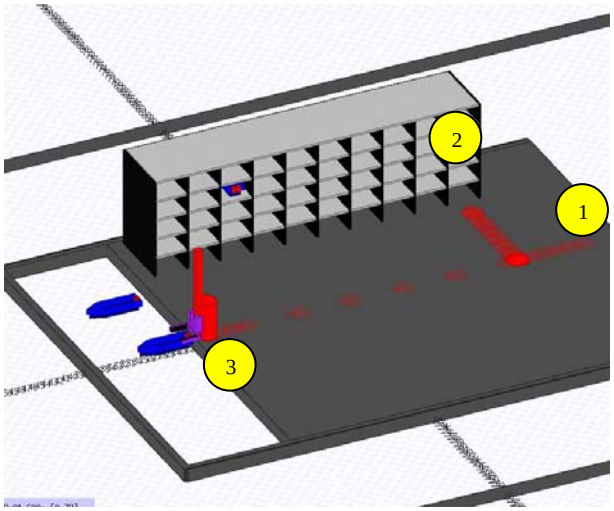

Figure 17: Marina simulation

8.2. The indoor marina

The indoor marina world simulates an indoor harbor and a lifter bot, the track of the lifter bot is shown in Figure 17. The client specifies the slot where the boat is parked, and the lifter bot moves to the boat and picks it up before moving to the drop point and putting the boat in the water.

The indoor marina world tests all the functions of the gripper and it is also a demonstration of a robotics solution to the marina management system.

8.3. The stacking test

To test whether the improved Stage has met the requirement for stacking objects, a world model containing small boxes was created. The tests show that the robot is able to pick up an object successfully and drop it at another location. However, because Stage does not have a physics engine and it only simulates simple interactions between objects, the objects being released by the gripper in the air will stay in the air. There are no concepts such as gravity and mass.

8.4. Test results

Each test is carried out in 3 different computers to ensure the improved version of Stage is portable.

The tests were all successfully completed. The tests' results suggest that the collision detection is working as intended and the gripper is functioning properly. However, the gripper device uses a very inefficient and error-prone method to update itself and perform its tasks.

9. Future work

A special kind of model called 'actuator' is added to the Stage project during the testing stage of our project; an actuator is a model where the blocks can be moved efficiently and safely. Merging the gripper model with the actuator model would greatly improve the robustness of the gripper model.

In the existing Stage, models are specified in a plain text file where the user has to specify all the vertices on the model and this can be a very time consuming task. A model editor that allows the user to draw the shape of the model and outputs the model's vertices in Stage's format would greatly reduce the time needed to construct a model.

10. Conclusions

The 2.5D Stage robotics simulator project has made several significant improvements to the Stage system. The collision detection function is modified to match the 2.5D Stage system. It can now accurately detect and signal a collision in a 2.5D environment. A gripper class is added to the system to perform tasks such as stacking objects on top of one another. The gripper class created is not only capable of closing on an object and carrying it around, but also allows the gripper to move vertically.

Tests performed by the improved system have shown satisfactory results and the patches are submitted to the Player/Stage open source project.

11. References

CLOC. SourceForge.Net. *Cloc*. Retrieved Aug 20, 2008 from: http://sourceforge.net/projects/cloc/

Gerkey, B. P., Vaughan, R. T. and Howard. A., ``The player/stage project: Tools for multi-robot and distributed sensor systems." In Proceedings of the International Conference on Advanced Robotics, Coimbra, Portugal, 2003.

Paul S. Heckbert (2004). *Graphics gems IV*. Boston: AP Professional. p. 366-369.

Koenig, N. and Howard, A., "Intelligent Robots and Systems" 2004. In Proceeding of IEEE/RSJ International Conference 2004.