PA3.1. PA3 (2021W2) - Image Partitioning Trees

Due: Monday, March 28, 22:00

Goals and Overview

In this PA (Programming Assignment) you will:

- learn about representing images using a binary tree
- learn to design more complex recursive algorithms
- learn about a possible method of image compression using space-partitioning trees

Part 1: The `PTree` Class

A `PTree` is a binary tree whose nodes represent rectangular regions of a PNG image. The root represents the entire image. The two children (referred to as `A` and `B`), of any node `nd` represent two equal or roughly equal horizontal or vertical partitions of `nd`'s image region. Every node also contains:

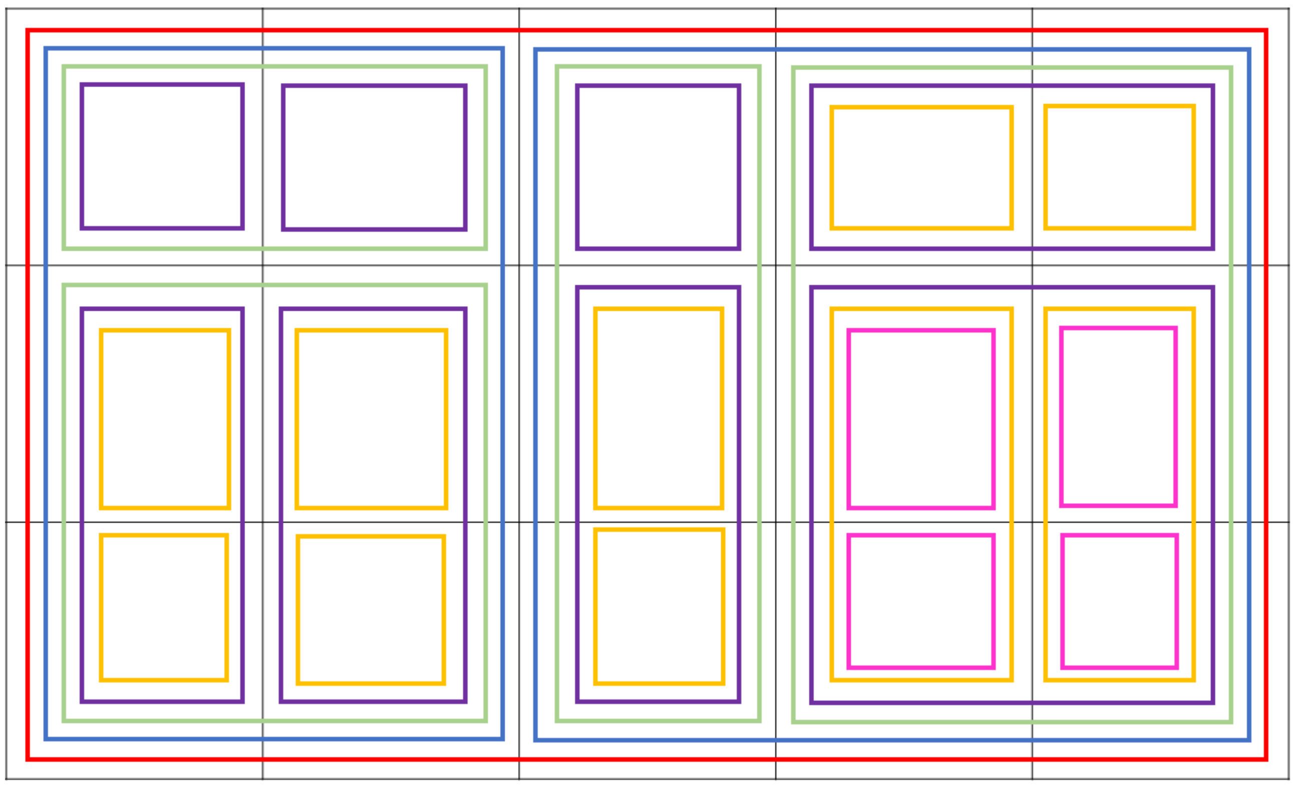
- a pixel representing the average colour of the pixels in its rectangular region
- the upper-left image coordinates of its rectangular region
- the width of its rectangular region in pixels, and
- the height of its rectangular region in pixels

Building a `PTree`

The contructor of the `PTree` receives a PNG image of non-zero dimensions.

The region for each node is split as equally as possible, and symmetrically, into two smaller regions along the longer of the two dimensions of the node being split. See the documentation in `ptree.h` for details about how to determine the split axis and how to determine the dimensions and coordinates of the split pieces. A freshly constructed `PTree` will have a leaf node corresponding to each individual pixel of the original image.

For example, a 5x3 image will be progressively split into the regions shown in the image below:



which has the corresponding tree structure below:

Programming Assignment 3

Assessment overview

Total points:	19/20
Score:	95%

Question

Value:	20
History:	19
Awarded points:	19/20

Report an error in this question

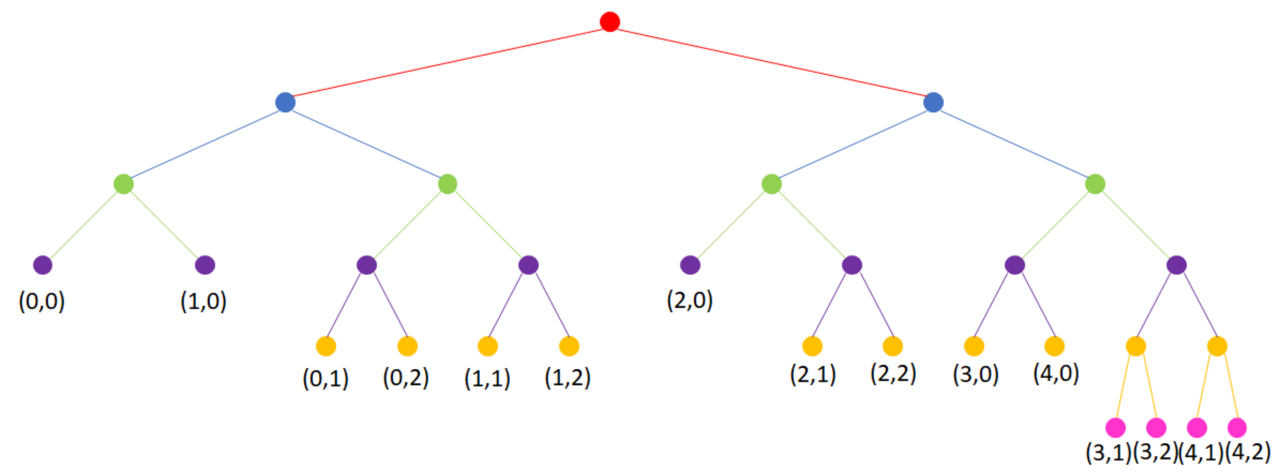
Previous question

Next question

Attached files

No attached files

Attachments can't be added or deleted because the assessment is closed.



Note that due to required symmetry of splitting, every node in the tree has exactly 0, or 2 children (i.e. full); there will never be a node with only one child.

To complete this part of the assignment, you should complete (or modify) the following functions:

In `class PTree`:

- `PTree(PNG& imIn)`
- `PTree(const PTree& other)`
- `PTree& operator=(const PTree& other)`
- `~PTree()`
- `void Clear()`
- `void Copy(const PTree& other)`
- `PNG Render() const`
- `int Size() const`
- `int NumLeaves() const`
- `Node* BuildNode(PNG& im, pair<unsigned int, unsigned int> ul, unsigned int w, unsigned int h)`
- `void FlipHorizontal()`
- `void FlipVertical()`

Many of the functions above will require private recursive helpers. It is up to you to add the declarations into `ptree-private.h`, giving them appropriate signatures, and implementing them in `ptree.cpp`.

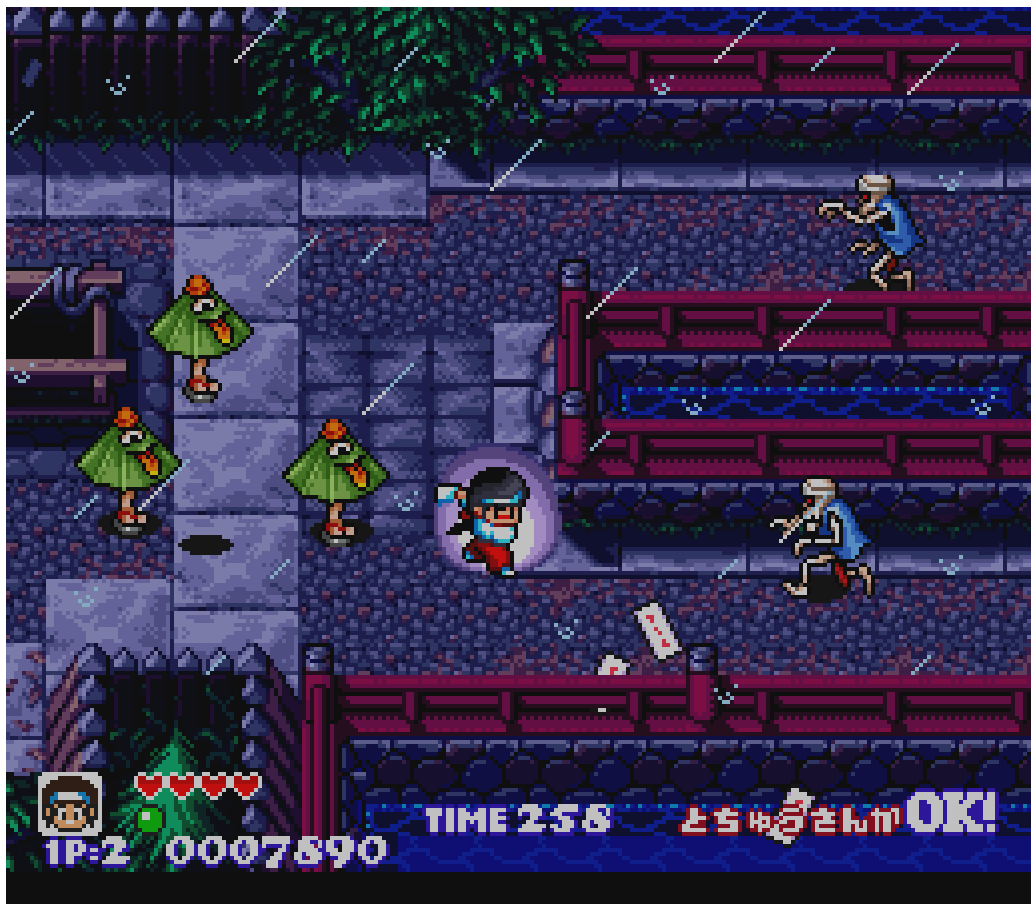
Advice: The constructor (which will use `BuildNode`) is critical for all of the other tree functionality. It is recommended to focus your efforts into first correctly implementing `BuildNode` and `Render` which are used in most of the testing functions.

Part 2: Image compression using `PTree`s

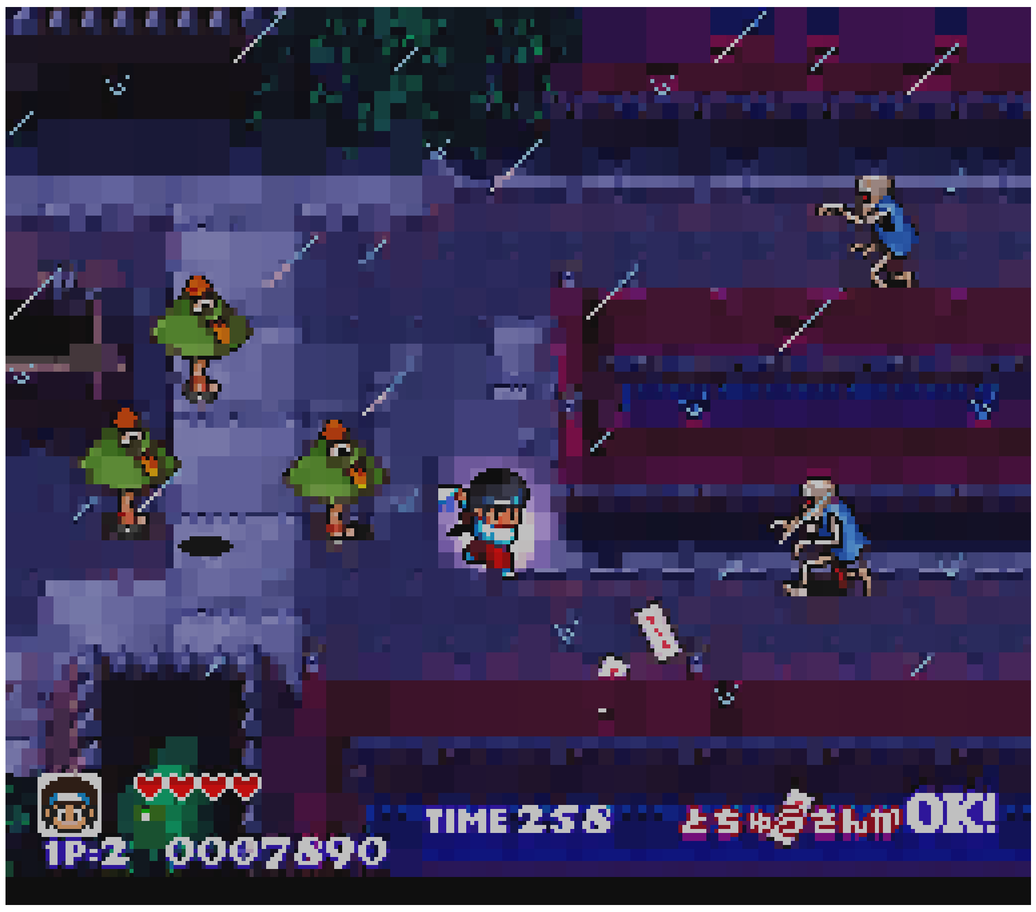
As a result of the hierarchical structure of the tree, along with each node storing the average colour of its rectangular region, we can trim away portions of the tree representing areas without fine pixel-level detail. This is achieved by the `Prune` function, which receives a tolerance parameter. This function attempts, starting near the top of a freshly built tree, to remove all of the descendants of a node, if all of the leaf nodes below the current node have colour within tolerance of the node's average colour.

In this way, areas of the image with little colour variability can be replaced with a single rectangle of solid colour, while areas with fine pixel detail can remain detailed. The image quality may be reduced, but fine details will still be visible, and the size of the structure used to store the image data will also be reduced. An example of the pruning effect is demonstrated in the pair of images below.

Original image (upscaled):



Rendered image after pruning at tolerance 0.05 (upscaled):



The tree constructed from the original image contains 57,344 leaf nodes; one for each pixel in the image.

The tree obtained after pruning contains 6,518 leaf nodes.

To complete this part of the assignment, you should complete (or modify) the following functions:

In `class PTree`:

- `void Prune(double tolerance)`

It may be helpful to write *two* recursive helpers for this function.

Testing

We have provided a test file `testPTree.cpp` which includes a basic set of tests. It is highly recommended to add to or modify the existing tests, especially to add tests which run your functions on *tiny* `PNG` images. The test executable can be compiled by typing:

```
make
```

Grading Information

The following files are used to grade this PA:

- `ptree-private.h`

- `ptree.cpp`

All other files (including any testing files you create) will not be used for grading.

Getting the Given Code

Download the source files here pa3-20220309-0028.zip and follow the procedures you learned in lab to move them to your home directory on the remote linux machines.

Submission

Correct answer

Submitted answer 21 **partially correct: 95%**

Submitted at 2022-03-27 12:20:54 (PDT)

hide

Files

ptree.cpp
uploaded

Download Show preview

ptree-private.h
uploaded

Download Show preview

Score: 19/20 (95%)

Output

```
Test group ptree output:
stdout: clang++ -stdlib=libc++ -std=c++1y -c -g -O0 -Wall -Wextra -pedantic
cs221util/PNG.cpp
clang++ -stdlib=libc++ -std=c++1y -c -g -O0 -Wall -Wextra -pedantic
cs221util/HSLAPixel.cpp
clang++ -stdlib=libc++ -std=c++1y -c -g -O0 -Wall -Wextra -pedantic
cs221util/lodepng/lodepng.cpp
clang++ -stdlib=libc++ -std=c++1y -c -g -O0 -Wall -Wextra -pedantic hue_utils.cpp
clang++ -stdlib=libc++ -std=c++1y -c -g -O0 -Wall -Wextra -pedantic testPTree.cpp
clang++ -stdlib=libc++ -std=c++1y -c -g -O0 -Wall -Wextra -pedantic -Wfloat-
conversion ptree.cpp
clang++ PNG.o HSLAPixel.o lodepng.o hue_utils.o testPTree.o ptree.o -stdlib=libc++
-std=c++1y -lc++abi -lpthread -lm -o testPTree

stderr:
Test group memtest output:
stdout: clang++ -stdlib=libc++ -std=c++1y -c -g -O0 -Wall -Wextra -pedantic
cs221util/PNG.cpp
clang++ -stdlib=libc++ -std=c++1y -c -g -O0 -Wall -Wextra -pedantic
cs221util/HSLAPixel.cpp
clang++ -stdlib=libc++ -std=c++1y -c -g -O0 -Wall -Wextra -pedantic
cs221util/lodepng/lodepng.cpp
clang++ -stdlib=libc++ -std=c++1y -c -g -O0 -Wall -Wextra -pedantic hue_utils.cpp
clang++ -stdlib=libc++ -std=c++1y -c -g -O0 -Wall -Wextra -pedantic testPTree.cpp
clang++ -stdlib=libc++ -std=c++1y -c -g -O0 -Wall -Wextra -pedantic -Wfloat-
conversion ptree.cpp
clang++ PNG.o HSLAPixel.o lodepng.o hue_utils.o testPTree.o ptree.o -stdlib=libc++
-std=c++1y -lc++abi -lpthread -lm -o testPTree

stderr:
```

Test Results

✓ **[3/3] PTree::Constructor**

✓ **[1/1] PTree::Size_NumLeaves_small**

✓ **[1/1] PTree::Size_NumLeaves_large**

✓ **[1/1] PTree::Copy_constructor**

✓ **[1/1] PTree::operator=_self**

✓ **[1/1] PTree::operator=_other**

✓ **[1/1] PTree::Render_small**

✓ **[2/2] PTree::Render_large**

✓ **[1/1] PTree::Prune_small_render**

✓ **[2/2] PTree::Prune_large_render**

✓ **[1/1] PTree::FlipHorizontal_small**

✓ **[1/1] PTree::FlipHorizontal_large**

✓ **[1/1] PTree::FlipVertical_small**

✓ **[1/1] PTree::FlipVertical_large**

✓ **[1/1] PTree::Prune_FlipH_FlipV**

✗ **[0/1] [memory] testPTree**

Submitted answer 20 **partially correct: 95%**
Submitted at 2022-03-27 12:11:16 (PDT)
show

Submitted answer 19 **partially correct: 95%**
Submitted at 2022-03-27 12:07:41 (PDT)
show

Show/hide older submissions