Linear interpolation

Gagarine Yaikhom

I.

We implement piece-wise linear interpolation.

⟨*Macro defintions*⟩≡

```
typedef enum {
    SUCCESS = 0,

    INVALID_POINT,
    INVALID_POINTS,
    INVALID_FIRST_POINT,
    INVALID_SECOND_POINT,
    INVALID_INTERPOLANTS,
    INVALID_NUMBER_OF_INTERPOLANTS,
    POINT_BELOW_LOWEST_INTERPOLANT,
    POINT_ABOVE_UPPERMOST_INTERPOLANT,
    POINT_IS_INTERPOLANT_POINT,

    NUM_ERRORS
}  error_code_t;
```

⟨*Data structures*⟩≡

```
typedef struct {
    float x, y;
} point2f_t;
```

⟨*Functions*⟩≡

```
void lerp_points_nocheck(
    point2f_t* points,
    size_t num_points,
    const point2f_t* a,
    const point2f_t* b
)
{
    /* Setup interpolant coefficients */
    float x1_x0 = b->x - a->x;
    float y1_y0 = b->y - a->y;
    float y0x1_y1x0 = a->y * b->x - b->y * a->x;
    float one_over_x1_x0 = 1.0F / x1_x0;

    for (size_t i = 0; i < num_points; ++i) {
        float y = points[i].x * y1_y0 + y0x1_y1x0;
        y *= one_over_x1_x0;
        points[i].y = y;
    }
}
```

⟨*Functions*⟩+≡

```
int lerp_points(
    point2f_t* points,
    size_t num_points,
    const point2f_t* a,
    const point2f_t* b
)
{
    if (points == NULL) {
        return INVALID_POINTS;
    }
    if (a == NULL) {
        return INVALID_FIRST_POINT;
    }
    if (b == NULL) {
        return INVALID_SECOND_POINT;
    }
    lerp_points_nocheck(points, num_points, a, b);
    return SUCCESS;
}
```

⟨*Functions*⟩+≡

```
int lerp_point(
    point2f_t* point,
    const point2f_t* a,
    const point2f_t* b
)
{
    if (point == NULL) {
        return INVALID_POINT;
    }
    if (a == NULL) {
        return INVALID_FIRST_POINT;
    }
    if (b == NULL) {
        return INVALID_SECOND_POINT;
    }
    lerp_points_nocheck(point, 1, a, b);
    return SUCCESS;
}
```

⟨*Functions*⟩+≡

```
bool floats_are_equal(
    float a,
    float b,
    float epsilon
)
{
    return fabs(a - b) < fabs(a) * epsilon;
}
```

⟨*Functions*⟩+≡

```
int find_interpolant_nocheck(
    const point2f_t** a,
    const point2f_t** b,
    point2f_t* point,
    const point2f_t* interpolants,
    size_t num_interpolants
)
{
    size_t begin = 0;
    size_t end = num_interpolants + 1;
    while (end - begin > 1) {
        size_t middle = (size_t)
            floor(0.5F * (end + begin));
        if (point->x < interpolants[middle].x) {
            end = middle;
        } else {
            begin = middle;
        }
    }
    *a = &(interpolants[begin]);
    *b = &(interpolants[end]);
    return SUCCESS;
}
```

⟨*Functions*⟩+≡

```
int point_inside_interpolant_nocheck(
    point2f_t* point,
    const point2f_t* interpolants,
    size_t num_interpolants
)
{
    if (point->x < interpolants[0].x) {
        return POINT_BELOW_LOWEST_INTERPOLANT;
    }
    if (point->x > interpolants[num_interpolants].x) {
        return POINT_ABOVE_UPPERMOST_INTERPOLANT;
    }
    return SUCCESS;
}
```

⟨*Functions*⟩+≡

```
int piecewise_lerp_points_nocheck(
    point2f_t* points,
    size_t num_points,
    const point2f_t* interpolants,
    size_t num_interpolants,
    error_code_t* error_codes
)
{
    int has_errors = false;
    for (size_t i = 0; i < num_points; ++i) {
        int is_inside =
            point_inside_interpolant_nocheck(
                &points[i], interpolants,
                num_interpolants);
        if (is_inside != SUCCESS) {
            if (error_codes != NULL) {
                error_codes[i] = is_inside;
            }
            has_errors = true;
            continue;
        }
        const point2f_t* a;
        const point2f_t* b;
        int r = find_interpolant_nocheck(
            &a, &b, &points[i],
            interpolants, num_interpolants);
        if (r == SUCCESS) {
            lerp_points_nocheck(
                &points[i], 1, a, b);
        }
        if (error_codes != NULL) {
            error_codes[i] = SUCCESS;
        }
    }
    return has_errors;
}
```

⟨*Functions*⟩+≡
```
int piecewise_lerp_points(
    point2f_t* points,
    size_t num_points,
    const point2f_t* interpolants,
    size_t num_interpolants,
    error_code_t* error_codes
)
{
    if (num_points < 1) {
        return SUCCESS;
    }
    if (points == NULL) {
        return INVALID_POINTS;
    }
    if (interpolants == NULL) {
        return INVALID_INTERPOLANTS;
    }
    if (num_interpolants < 1) {
        return INVALID_NUMBER_OF_INTERPOLANTS;
    }
    int r = piecewise_lerp_points_nocheck(
        points, num_points, interpolants,
        num_interpolants, error_codes);
    return r;
}
```

⟨*Functions*⟩+≡
```
void print_point(
    const point2f_t* point,
    const char* title
)
{
    if (title != NULL) {
        printf("%s", title);
    }
    printf("(%+f, %+f)\n",
           point->x, point->y);
}
```

⟨*Functions*⟩+≡
```
void print_points(
    point2f_t* points,
    size_t num_points,
    const char* title
)
{
    if (title != NULL) {
        printf("%s", title);
    }
    for (size_t i = 0; i < num_points; ++i) {
        print_point(&points[i], "\t");
    }
}
```

⟨*Test functions*⟩≡
```
int test_piecewise_lerp()
{
    size_t num_points = 16;
    point2f_t points[16];
    error_code_t errors[16];
    float v = -6.0F;
    for (size_t i = 0; i < num_points; ++i) {
        points[i].x = v;
        points[i].y = 0.0F;
        v += 1.0F;
    }

    size_t num_interpolants = 5;
    point2f_t interpolants[] = {
        {-5.0F, 0.0F}, {-2.0F, 3.0F}, {0.0F, 3.0F},
        { 2.0F, 1.0F}, { 5.0F, 4.0F}, {8.0F, 1.0F},
    };
    piecewise_lerp_points(
        points, num_points, interpolants,
        num_interpolants, errors);
    printf("Error codes:\n");
    for (size_t i = 0; i < num_points; ++i) {
        printf("\t[%3lu] %d\n", i, errors[i]);
    }
    print_points(points, num_points,
        "test_piecewise_lerp_points():\n");
    return SUCCESS;
}
```

II. Main program

Main program for testing the functions.

⟨*Main*⟩≡
```
int main(void) {
    int r = test_piecewise_lerp();
    if (r != SUCCESS) {
        fprintf(stderr, "test_piecewise_lerp() failed!\n");
    }
    return r;
}
```

⟨*Standard libraries*⟩≡
```
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
```

⟨*Function declarations*⟩≡
```
void print_point(const point2f_t*, const char*);
```

⟨*algorithm.c*⟩≡
⟨*Standard libraries*⟩
⟨*Macro defintions*⟩
⟨*Data structures*⟩
⟨*Function declarations*⟩
⟨*Functions*⟩
⟨*Test functions*⟩
⟨*Main*⟩