Real Fake DOMs: Verified Browser Rendering

Jeffrey Cai
jeffreycai@college.harvard.edu

Jason Goodman
jgoodman@college.harvard.edu

ABSTRACT

In this writeup we present Real Fake DOMs, the first Coq development to our knowledge to model Document Object Models or the graphics primitives necessary to model browser rendering algorithms in a shallow embedding.

As a first foray into verifying algorithms in this area, we model simplified DOMs with representative background-color, width, height, position, top, left, and overflow attributes. Additional features could in theory be implemented in terms of these, though it would be impractical to support a richer set of features in an exploratory development.

Building on these models, we define and verify a series of renderers inspired by basic optimiations used in web browsers. We prove correctness by showing functional equivalence to an unoptimized reference renderer. Perhaps our biggest takeaway is a reminder that straightforward facts can require immense effort to prove if algorithms are not designed with verification in mind.

1. INTRODUCTION

As web applications have grown more sophisticated and browser optimizations more aggressive, browser rendering glitches have emerged as the consistent top concern in the Gallup "Most Important Problem" poll.[1] With this peeve as motivation, we set out to create mock renderers in Coq with optimizations that are proved not to affect their functional behavior.

The result of this effort is a model of graphics operations, a datatype for expressing a subset of DOM features, an unoptimized reference implementation of a DOM renderer, a renderer that paints directly onto a single graphic without compositing and skips out-of-bounds elements, and renderer that groups elements into layers for memoizable rendering in the common case of local DOM updates. Admittedly, this last renderer's correctness proof contains an unproven arithmetic fact that we believe to be true but beyond our late-night abilities.

Because our Coq programs can only generate textual output, we have additionally created a exporter that transfers rendered graphics to an HTML canvas. We have not verified that this exporter or our browsers are correct.

[1]Not actually true, with high probability.

2. MODELING GRAPHICS

It was not immediately clear how to model graphics in Coq. At a high level, our first choice was between mapping pixel coordinates to color values—simulating arrays—or modeling sequences of primitive drawing commands, which might be a more direct representation of a browser renderer's output.

We are confident that the pixel model is preferable, given that the sense of equivalence we sought concerns exactly the mapping from coordinates to colors. Primitive drawing operations, such as painting boxes, are easy to implement using this model, and the alternative would require a manual definition of equivalence—likely defining a mapping to pixels—in the trusted computing base.

An additional design decision with our graphics model was the choice of data structure used to represent pixel arrays. Given that our goal is to model rather than implement data structures used in practice, we primarily sought out expressiveness and ease of programming. To motivate our claims of tedium, consider the seemingly straightforward `composite` operation that stamps one graphic onto another, deferring to the original graphic where the new one is transparent.

Lists of lists are perhaps the closest conceptual analog to C arrays in the Coq library, but manipulating them would be tedious. Compositing and similar operations would require recursion on their arguments, necessitating inductive proofs of basic properties and potentially bug-laden handling of graphics with different dimensions. Fatally, lists do not naturally support pixels in negative coordinates, a feature that is not used by ac-

tual renderers but which we found to be useful in specifying a straightforward reference renderer.

Perhaps the most efficient option, a tree accessed through Coq's finite map interface, would allow for negative coordinates and make compositing outside of a graphic's bounds easier to reason about. However, the remaining problems with lists still apply.

For this reason, we chose to model graphics as Coq functions from unbounded integer coordinates to colors, adding the axiom of functional extensionality so that equivalent graphics created by different algorithms can be satisfyingly claimed equal. We did not model alpha values but included a "None" color for use outside of a graphic's bounds or for transparency.

Intriguingly, our reliance on a small set of functions for compositing, clipping, and box drawing means we have effectively modeled traces of primitive operations in a structure of closures.

One abandoned idea was to model graphics of fixed size using dependent types. This proved to require significantly more effort than writing proofs to reason about the bounds used by individual functions, and we found that the size of a graphic is rarely of interest for our purposes.

```
Definition composite base overlay offset : graphic
    := fun coord =>
  match offset, coord with
  | Coord x y, Coord x' y' =>
    let overlay_color := overlay (Coord (-x + x')
        (-y + y')) in
    match overlay_color with
    | None_c => base coord
    | _ => overlay_color
    end
  end.
```

Listing 1: The composite function

3. MODELING DOMS

We aimed to model a number of DOM attributes at a sufficient level of detail to capture the spirit of web rendering and optimizations without creating too big of an undertaking. This section contains brief descriptions of the supported fields, which are interpreted according to the CSS specification, as well as a discussion of our inductive data structure.

3.1 Attributes

3.1.1 Position

Elements can be positioned in vertical (block) **static** layout, **relative** to their static positions, or in **absolute** coordinates from the origin of the closest non-static ancestor. Non-static coordinates are in pixels from the top and left.

Note that in general, a child or later sibling is drawn on top of earlier elements in the DOM, but non-static elements are drawn above their associated layers of statically-positioned elements.

3.1.2 Width and height

Every element has specified dimensions in pixels. An interesting extension of our work would be to add horizontal layouts and compute heights dynamically.

3.1.3 Overflow

Elements can have overflow either visible or hidden. If the overflow is hidden, then renderings of descendants are clipped outside of the element's width and height.

3.1.4 Background color

Every element has an RGB or "none" background.

3.2 Data Structure

Our initial attempt to model DOMs in Coq had each node contain a list of children and an optional color. This followed from an intuitive conception of DOMs and seemed to allow for the use of map, fold, and other library functions in implementing algorithms.

Unfortunately, Coq does not automatically support induction and recursion with this use of types, so we resorted instead to having an element "contain" its child and next sibling. As a side-effect, our recursive algorithms mostly operate not only on an element and its subtree, but its siblings' subtrees as well.

```
Inductive position := Static | Relative | Absolute.
Inductive overflow := Visible | Hidden.
Inductive attributes :=
  Attributes : forall
    (left top width height : Z)
    (color : color)
    (pos : position)
    (ovf : overflow),
    attributes.
Inductive dom : Set :=
| Dom : attributes -> forall (child sibling : dom),
     dom
| None_d.
```

Listing 2: The DOM model

4. PROOF STRATEGY

4.1 Overview

We defined a straightforward reference renderer in terms of simple graphics operations (draw box and composite) and a "dumb" inefficient recursion that would translate to creating a new graphic for each DOM element.

After proving facts to act as unit tests, and hours of manual testing—the state of the art in browser development—we deemed the reference render "trusted." We then proved that our optimized renderers produce the same renderings as this one.

4.2 Structure

We essentially used two layers in our proofs:

1. Prove lemmas concerning properties of different graphics operations: for instance, that compositing the blank graphic onto a graphic leaves it unchanged, or that translating two rectangles translates their intersection without changing the dimensions.

2. Prove the equivalence of the renderers by applying the lemmas.

This has a twofold advantage: We are not bogged down with "trivial" goals while dealing with more complicated proof strategies, and the two models are decoupled to the extent that either could be swapped out in a continuation of our work.

4.3 Graphics Proofs

One proof tactic we used was `extensionality`: when faced with a goal of proving that two graphics were equal, we could simply prove that the rendered color was equal at every test point. This allowed us to make progress by unfolding the definitions of graphics primitives, which are written in terms of what they do to pixels.

Another common theme among graphics proofs was the use of `Z` (the module implementing integers) and `ring` tactics. Since there was a lot of arithmetic and inequalities involved in computing bounds, we quickly became adept at manipulating integers in various ways.

Listing 3: An example of a graphics lemma: clipping distributes over compositing. Note the use of `extensionality` and `ring` tactics.

```
Lemma clip_composite_distr g1 g2 offset pos dim:
  clip (g1 CC g2 @ offset) pos dim =
  (clip g1 pos dim) CC (clip g2 (subtr_c pos
      offset) dim) @ offset.
Proof.
  extensionality x.
  destruct x as [xt yt], pos as [x y], dim as [w
      h], offset as [x' y'].
  simpl.
  repeat destruct in_box_dec; auto.
  - apply (in_box_shift _ _ _ _ _ _ (-x') (-y')) in
      i.
    ring_simplify (x + - x') in i.
    ring_simplify (y + - y') in i.
    replace (xt + - x') with (- x' + xt) in i by
        ring.
    replace (yt + - y') with (- y' + yt) in i by
        ring.
    contradiction.
  - apply (in_box_shift _ _ _ _ _ _ x' y') in i.
    ring_simplify in i.
    ring_simplify (x - x' + x') (y - y' + y') in i.
    contradiction.
Qed.
```

4.4 Rendering Proofs

The first thing that we did when writing a rendering proof was to induct on the DOM (possibly generalizing some variables first), generating inductive hypotheses that we could apply to children and siblings. Once we had done that, the next step was generally to `destruct` the position-type (`Static | Relative | Absolute`) and the overflow type (`Visible | Hidden`), generating a variety of different cases. Usually by this point, everything could be fully expanded and we could see plainly what we were trying to prove equal in this case.

```
Definition magenta_square :=
  Dom (Attributes 5 5 10 10
    magenta Absolute Visible)
  None_d None_d.
Definition blue_block :=
  Dom (Attributes 0 0 30 30
    blue Static Hidden)
  magenta_square None_d.
Definition yellow_stripe :=
  Dom (Attributes 0 0 100 5
    yellow Static Visible)
  None_d blue_block.
Definition green_block :=
  Dom (Attributes 5 0 55 40
    green Relative Visible)
  yellow_stripe None_d.
Definition red_block :=
  Dom (Attributes 5 5 20 20
    red Static Hidden)
  yellow_stripe green_block.
Definition test :=
  Dom (Attributes 0 0 100 70
    black Absolute Hidden)
  red_block None_d.
```

Figure 1: A test example written in Coq and exported to the HTML renderer.

The next step was generally to fire a barrage of graphics lemmas in order to simplify and normalize the expressions. Unfortunately, much of the normalization had to be done manually (using `rewrite` and pinpointing specific expressions). Also, the work leftover from applying the graphics lemmas usually involved some `ring` arithmetic, which again had to be done manually. It is very likely that if we had the know-how to create sophisticated `Ltac`s to automate our proofs, they could be made much shorter.

5. RENDERERS

Here, we describe the three kinds of renderers that we implemented: the reference renderer, the incremental renderer, and the layering renderer. It was interesting to see firsthand the many different ways that graphics could be drawn, clipped, and composited together to produce equivalent end results.

5.1 Reference Renderer

Making the reference renderer ended up being quite awkward, primarily because of several technical details in the specification: for example, static elements should be rendered underneath non-static elements, and absolute positioning is relative to the position of the nearest non-static ancestor. Due to these issues, we split up the reference renderer into a `render0` function which rendered static elements only, then `render'` which used `render0` as a subroutine and rendered the rest of the elements.

Additionally, we performed some tests and sanity checks on the reference renderer. We created an HTML page

that would take a printed Coq structure and render the pixels. This was also awkward, however: our "testing" involved manually creating test DOMs in Coq and copy-pasting the exported structure to a browser. Furthermore, some of the properties noted above are subtle and bugs manifest only when nesting deeply with a variety of overflow and position settings.

To partially remedy this, we proved some lemmas about the reference renderer to act as sanity checks. For example, we showed that the `left` and `top` attributes for a `Static` element were irrelevant to the rendered result. We also showed that if a color exists in the rendered graphic, then it must exist in some DOM element.

```
Function render0 dom pos : graphic :=
  match dom, pos with
  | None_d, _ => blank
  | Dom (Attributes l t w h c p o) child sib, Coord
      x y =>
    match p with
    | Static =>
      blank CC (box (Dim w h) c) @ pos
            C0 (clip_ovf o pos (Dim w h) (render0
                 child pos))
            C0 (render0 sib (Coord x (y + h)))
    | Relative =>
      render0 sib (Coord x (y + h))
    | Absolute =>
      render0 sib pos
    end
  end.

Function render' dom pos : graphic :=
  match dom, pos with
  | None_d, _ => blank
  | Dom (Attributes l t w h c p o) child sib, Coord
      x y =>
    match p with
    | Static =>
      (clip_ovf o pos (Dim w h) (render' child pos))
        C0 (render' sib (Coord x (y + h)))
    | Relative =>
      let pos' := Coord (x + l) (y + t) in
      clip_ovf o pos' (Dim w h)
        (blank CC (box (Dim w h) c) @ pos'
               CC (render0 child c0) @ pos'
               CC (render' child c0) @ pos')
        C0 (render' sib (Coord x (y + h)))
    | Absolute =>
      let pos' := Coord l t in
      clip_ovf o pos' (Dim w h)
        (blank CC (box (Dim w h) c) @ pos'
               CC (render0 child c0) @ pos'
               CC (render' child c0) @ pos')
        C0 (render' sib pos)
    end
  end.
```

Listing 4: The reference renderer, defined first for static layers. C0 and CC are compositing operations.

5.2 Incremental Renderer

This optimization of the reference renderer moves in the opposite extreme by painting boxes onto a single graphic that is threaded through recursive function calls. Essentially, we never composite together two complex graphics or clip a complex graphic; we only ever paste boxes onto an accumulating graphic. To do so correctly requires not only another level of contextual arithmetic but local observation of hidden overflow boundaries higher in the element tree.

```
Function inc_render' dom pos cd g offset : graphic
    :=
  match dom, pos with
  | None_d, _ => g
  | Dom (Attributes l t w h c p o) child sib, Coord
      x y =>
    match p with
    (* ... snipped ... *)
    | Absolute => (* Do a static pass, then a
        positioned pass. *)
      let pos' := Coord l t in
      let (bg_pos, bg_dim) := clip_box cd (add_c
          pos' offset) (Dim w h) in
      let g := g C0 (box_at bg_pos bg_dim c) in
      (* Do a static pass, then a positioned pass.
          *)
      let child_cd := restrict_clip cd o (add_c
          pos' offset) (Dim w h) in
      let g := inc_render0 child c0 child_cd g
          (add_c pos' offset) in
      let g := inc_render' child c0 child_cd g
          (add_c pos' offset) in
      inc_render' sib pos cd g offset
```

Listing 5: A sample case in the incremental renderer.

The call `inc_render' dom pos cd g offset` indicates that the element `dom` is to be rendered at position `pos`, then clipped according to a clipping directive `cd` before being pasted onto a base graphic `g` at an offset `offset`. Note that `clip_box` is used in order to compute bounds for a clipped background box, to render the current DOM element; and `restrict_clip` is used to compute clipping bounds to be passed to the child subtree. Both functions are based on the primitive `box_intersect`, which uses a well-known formula involving `Z.max` and `Z.min` in order to compute the intersection of two rectangles.

5.3 Layering Renderer

This renderer takes a more conventional approach in painting the DOM in layers that group static descendents of positioned elements. This allows for the layer renderings to be retained and reused if the DOM is slightly modified in the future. Changes to non-static positions only require re-compositing layers at the correct offsets, while more disruptive changes still only require re-rendering affected layers.

It should be noted that our implementation does not "memoize" the result of rendering a layer, although we did attempt to syntactically mirror the idea by extending the layer rendering function with particular values that had been computed. We abandoned the idea because threading this non-inductive output through the renderer broke our correctness proof.

```
Function paint_layer d :=
```

```
  match d with
  | None_d => blank
  | Dom (Attributes _ _ w h c p o) child _ =>
      let g := box (Dim w h) c in
      inc_render0 (normalize child) c0 Don't_clip g
          c0
  end.

Function layer_render dom pos cd g offset : graphic
     :=
  match dom, pos with
  | None_d, _ => g
  | Dom (Attributes l t w h c p o) child sib, Coord
       x y =>
    match p with
    | Static =>
      let child_cd := restrict_clip cd o (add_c pos
           offset) (Dim w h) in
      let g := layer_render child pos child_cd g
           offset in
      layer_render sib (Coord x (y + h)) cd g offset
    | Relative =>
      let pos' := Coord (x + l) (y + t) in
      let child_cd := restrict_clip cd o (add_c
           pos' offset) (Dim w h) in
      let g_child := paint_layer dom in
      let g_child := blank CC g_child @ (add_c pos'
           offset) in
      let g_child := restrict_clip_g child_cd
           g_child in
      let g := g C0 g_child in
      let g := layer_render child c0 child_cd g
           (add_c pos' offset) in
      layer_render sib (Coord x (y + h)) cd g offset
    | Absolute =>
      let pos' := Coord l t in
      let child_cd := restrict_clip cd o (add_c
           pos' offset) (Dim w h) in
      let g_child := paint_layer dom in
      let g_child := blank CC g_child @ (add_c pos'
           offset) in
      let g_child := restrict_clip_g child_cd
           g_child in
      let g := g C0 g_child in
      let g := layer_render child c0 child_cd g
           (add_c pos' offset) in
      layer_render sib pos cd g offset
    end
  end.
```

Listing 6: The layer renderer, first defined in terms of rendering a single static layer with the incremental renderer

6. PROOF WALKTHROUGHS

In order to demonstrate what the proof structure is like, as well as to share a more detailed look at what's going on in our Coq development, we provide a moderately-detailed look at some sample cases in our proofs.

6.1 A graphics lemma

We step through the proof of a simple graphics lemma, `clip_composite_distr`:

```
(* Clipping distributes over compositing. *)
Lemma clip_composite_distr g1 g2 offset pos dim:
  clip (g1 CC g2 @ offset) pos dim =
  (clip g1 pos dim) CC (clip g2 (subtr_c pos
       offset) dim) @ offset.
```

First, we notice that we are trying to prove two graphics equal, so the natural first tactic is `extensionality`. Then, in order to get everything to simplify, we `destruct` all of the coordinates and dimensions.

```
  extensionality x.
  destruct x as [xt yt], pos as [x y], dim as [w
       h], offset as [x' y'].
  simpl.
```

At this point, we have unfolded `clip` into a series of `in_box_dec` computations. The only way to proceed now is to `destruct` them.

```
  repeat destruct in_box_dec; auto.
```

Now we have two cases; they are essentially the same, so we only consider the first. The goal does not matter, as we have two contradicting hypotheses:

```
i : in_box x y w h xt yt
n : ~ in_box (x - x') (y - y') w h (- x' + xt) (-
     y' + yt)
```

We apply the lemma `in_box_shift` and some ring simplifications to derive a contradiction.

```
  - apply (in_box_shift _ _ _ _ _ _ (-x') (-y')) in
      i.
    ring_simplify (x + - x') in i.
    ring_simplify (y + - y') in i.
    replace (xt + - x') with (- x' + xt) in i by
        ring.
    replace (yt + - y') with (- y' + yt) in i by
        ring.
    contradiction.
```

6.2 A render equivalence proof

Here, we step through a section of the main proof that `inc_render` is equivalent to the reference renderer.

```
Lemma inc_render'_equiv d pos cd g offset:
  inc_render' d pos cd g offset =
  g CC apply_clip (translate_clip cd offset)
       (inc_render' d pos Don't_clip blank c0) @
       offset.
```

Listing 7: An essential lemma for `inc_render`'s equivalence proof.

Instead of proving that `inc_render` is equivalent to `render` directly, we found it easier to first prove a semantic claim about `inc_render` that we called "paste equivalence." Essentially, the lemma statement says that calling `inc_render'` is equivalent to rendering on top of a blank graphic, clipping the result, and pasting onto the base graphic. The paste equivalence lemma bore the brunt of the proof difficulty, and the actual correctness theorem was fairly short by comparison. Hence, we step through a section of the paste equivalence lemma now.

The example is the `Relative` position case. The goal we have is the following:

```
(let (bg_pos, bg_dim) :=
   clip_box cd (Coord (x + l + x') (y + t + y'))
       (Dim w h) in
 inc_render' d2 (Coord x (y + h)) cd
   (inc_render' d1 c0
      (restrict_clip cd o (Coord (x + l + x') (y +
          t + y')) (Dim w h))
      (inc_render0 d1 c0
         (restrict_clip cd o (Coord (x + l + x') (y
             + t + y')) (Dim w h))
         (g C0 box_at bg_pos bg_dim c) (Coord (x + l
             + x') (y + t + y')))
      (Coord (x + l + x') (y + t + y'))) (Coord x'
          y')) =
composite g
  (apply_clip (translate_clip cd (Coord x' y'))
    (inc_render' d2 (Coord x (y + h)) Don't_clip
      (inc_render' d1 c0
         (restrict_clip Don't_clip o (Coord (x + l
             + 0) (y + t + 0))
            (Dim w h))
         (inc_render0 d1 c0
            (restrict_clip Don't_clip o (Coord (x
                + l + 0) (y + t + 0))
              (Dim w h))
            (blank C0 box_at (Coord (x + l + 0) (y
                + t + 0)) (Dim w h) c)
            (Coord (x + l + 0) (y + t + 0)))
         (Coord (x + l + 0) (y + t + 0))) c0))
           (Coord x' y')
```

First, to get rid of the annoying `let...in` syntax, we use a `remember` tactic.

```
 - remember (clip_box cd (Coord (x + l + x') (y +
     t + y')) (Dim w h)) as bg;
   destruct bg as [bg_pos bg_dim].
```

Then we apply several simplifications. `composite_onto_blank` says that compositing onto a blank graphic at the zero offset is a no-op. The inductive hypotheses `IHd1, IHd2` allow us to normalize calls to `inc_render'`, replacing the clip-directive parameter and the base graphic with concrete clips and concrete compositing onto a base graphic, respectively. We have to apply `IHd1` with explicit arguments to pinpoint what we want to rewrite, since `rewrite IHd1` after the first would uselessly re-expand when the clip-directive is already `Don't_clip` and the base graphic is already `blank`.

```
unfold composite0.
rewrite composite_onto_blank.
ring_simplify (x + l + 0) (y + t + 0).
rewrite IHd1.
rewrite IHd2.
rewrite inc_render0_equiv.
rewrite (IHd1 _ (restrict_clip Don't_clip o (Coord
    (x + l) (y + t))
          (Dim w h))).
rewrite (inc_render0_equiv _ _
  (restrict_clip Don't_clip o (Coord (x + l) (y +
     t)) (Dim w h))).
rewrite (IHd2 _ _ (_ CC _ @ _ CC _ @ _)).
```

At this point we can't go much further without splitting into case-analysis. We split into the cases where the clip-directive is `Don't_clip` versus `Clip_to`. We omit the former since it is strictly easier. Then, as a first step, we distribute `clip` over `composite` and normalize associativity for `composite`.

```
destruct cd as [|[cx cy] [cw ch]]; simpl.
+ (* -- Don't_clip case snipped -- *)
+ repeat rewrite clip_composite_distr.
  repeat rewrite composite_assoc.
```

The next step is to prove that the background boxes we render are the same on both sides. First, we rewrite using `clip_box_correct`, which states that clipping a box graphic gives a new box graphic at the right bounds computed by `box_intersect`. Then, we use a lemma, `box_intersect_shift`, to prove that the intersection of two translated boxes equals the translation of the intersection of the two original boxes.

```
  rewrite clip_box_correct.
  unfold clip_box in Heqbg.
  remember (box_intersect (Coord (cx - x') (cy -
      y')) (Dim cw ch)
    (Coord (x + l) (y + t)) (Dim w h)) as bg';
       destruct bg' as [bg_pos' bg_dim'].
  rewrite (composite_box_shift _ _ _ _ (Coord x'
      y')).
  rewrite add_c0_r, sub_c0.
  pose (box_intersect_shift
    (Coord cx cy) (Dim cw ch)
    (Coord (x + l + x') (y + t + y')) (Dim w h)
    (Coord (-x') (-y'))).
  rewrite <- Heqbg in e.
  unfold add_c at 1 2 in e.
  ring_simplify (cx + - x') (cy + - y')
    (x + l + x' + - x') (y + t + y' + - y') in e.
  rewrite <- Heqbg' in e.
  apply pair_eq in e; destruct e.
  rewrite H, H0.
  clear - IHd1 IHd2.
  destruct bg_pos; simpl.
  ring_simplify (x0 + - x' + x') (y0 + - y' + y').
```

With this step complete, we turn our attention to proving that the child graphic has the correct clipping bounds, which might be restricted depending on `overflow`. Hence, we split into `Visible | Hidden` cases, the former of which is fairly straightforward:

```
  destruct o;
    repeat rewrite clip_composite_distr;
    repeat rewrite composite_assoc.
  * simpl.
    (* -- ring simplifications, snipped -- *)
    auto.
```

Finally, the `Hidden` case; this is quite similar to the earlier section showing that the background boxes are the same on both sides, but using `nested_clip_correct` instead of `clip_box_correct`. Notice that `box_intersect_shift` is used in essentially the same way as before:

```
  * unfold restrict_clip, clip_intersect.
```

```
remember (box_intersect (Coord cx cy) _ _ _) as
    z;
  destruct z as [[cx1 cy1] [cw1 ch1]].
unfold translate_clip, apply_clip, subtr_c.
repeat rewrite nested_clip_correct.
remember (box_intersect (Coord (x + l - (x +
    l)) _) _ _ _) as z';
  destruct z' as [[cx2 cy2] [cw2 ch2]].
rewrite box_intersect_comm in Heqz'.
pose (box_intersect_shift (Coord (cx - x' - (x
    + l)) (cy - y' - (y + t)))
  (Dim cw ch) (Coord (x + l - (x + l)) (y + t -
      (y + t))) (Dim w h)
  (Coord (x + l + x') (y + t + y'))).
unfold add_c in e.
rewrite <- Heqz' in e.
(* -- ring simplifications, snipped -- *)
rewrite <- Heqz in e.
inversion_clear e.
clear - IHd1 IHd2.
(* -- ring simplifications, snipped -- *)
auto.
```

7. CONCLUSION/FUTURE WORK

7.1 Source Code

The source code for our Coq development is currently available at `https://github.com/jcai1/coq-dom`.

7.2 Local Differential Renderer

A planned renderer we were unable to complete, but within grasp of, is a differential renderer that not only renders in layers but conservatively identifies changes that can be made by repainting a subtree on top of a previous rendering of its layer. For example, if the only change to a layer is the solid background color of an element, that element's subtree can be redrawn instead of making an entirely new rendering of its layer.

7.3 Future Work

Some future directions might include:

- Supporting alpha and transparency
- Simplifying many of the proofs by creating `Ltac`s that automate some processes
- Proving more results about the reference renderer, or that it conforms to an explicit specification
- Generalizing DOM elements to other things than solid-colored boxes.