Light Template B/light

Purpose

The determination of light, visibility and physical access.

§**1. Darkness.** "Darkness" is not really a place: but in I6 it has to be an object so that the location-name on the status line can be "Darkness". In I7 we use it as little as possible: note that it has no properties.

```
Object thedark "(darkness object)";
```

§**2. Light Measurement.** These two routines, `OffersLight` and `HasLightSource`, are largely unchanged from their I6 definitions; see the *Inform Designer's Manual*, 4th edition, for a commentary. In terms of how they are used in I7, `OffersLight` is called only by the two rules below; `HasLightSource` is called also in determining the scope, that is, what is visible to the player.

```
[ OffersLight obj j;
    while (obj) {
        if (obj has light) rtrue;
        objectloop (j in obj) if (HasLightSource(j)) rtrue;
        if ((obj has container) && (obj hasnt open) && (obj hasnt transparent)) rfalse;
        if ((obj provides component_parent) && (obj.component_parent))
            obj = obj.component_parent;
        else
            obj = parent(obj);
    }
    rfalse;
];
[ HasLightSource i j ad sr po;
    if (i == 0) rfalse;
    if (i has light) rtrue;
    if ((IsSeeThrough(i)) && (~~(HidesLightSource(i))))
        objectloop (j in i)
            if (HasLightSource(j)) rtrue;
    ad = i.&add_to_scope;
    if (parent(i) ~= 0 && ad ~= 0) {
        if (metaclass(ad-->0) == Routine) {
            ats_hls = 0; ats_flag = 1;
            sr = scope_reason; po = parser_one;
            scope_reason = LOOPOVERSCOPE_REASON; parser_one = 0;
            RunRoutines(i, add_to_scope);
            scope_reason = sr; parser_one = po;
            ats_flag = 0; if (ats_hls == 1) rtrue;
        }
        else {
            for (j=0 : (WORDSIZE*j)<i.#add_to_scope : j++)
                if ((ad-->j) && (HasLightSource(ad-->j) == 1)) rtrue;
```

```
        }
    }
    if (ComponentHasLight(i)) rtrue;
    rfalse;
];
[ ComponentHasLight o obj next_obj;
    if (o provides component_child) {
        obj = o.component_child;
        while (obj) {
            next_obj = obj.component_sibling;
            if (obj has light) rtrue;
            if (HasLightSource(obj)) rtrue;
            if ((obj provides component_child) && (ComponentHasLight(obj))) rtrue;
            obj = next_obj;
        }
    }
    rfalse;
];
[ HidesLightSource obj;
    if (obj == player) rfalse;
    if (obj has transparent or supporter) rfalse;
    if (obj has animate) rfalse;
    if (obj has container) return (obj hasnt open);
    return (obj hasnt enterable);
];
```

§**3. Invariant.** The following routines maintain two variables about the light condition of the player:

(a) If on the most recent check the player was in light, then `location` equals `real_location` and `lightflag` is false.

(b) If on the most recent check the player was in darkness, then `location` equals `thedark` and `lightflag` is true.

Note that they are not allowed to alter `real_location`, whose definition has nothing to do with light.

```
Global lightflag = false;
```

§**4. Adjust Light Rule.** This rule fires at least once a turn, and more often when the player moves location, since that's likely to invalidate any previous assumptions. It compares the state of light now with the last time it ran, and gives instructions on what to do in each of the four possibilities.

```
[ ADJUST_LIGHT_R previous_light_condition;
    previous_light_condition = lightflag;
    lightflag = OffersLight(parent(player));
    if ((previous_light_condition == false) && (lightflag == false)) {
        location = thedark;
        rfalse;
    }
    if ((previous_light_condition == false) && (lightflag == true)) {
        location = real_location;
        CarryOutActivity(PRINTING_NEWS_OF_LIGHT_ACT);
        rfalse;
    }
    if ((previous_light_condition == true) && (lightflag == false)) {
        location = thedark;
        DivideParagraphPoint();
        BeginActivity(PRINTING_NEWS_OF_DARKNESS_ACT);
        if (ForActivity(PRINTING_NEWS_OF_DARKNESS_ACT) == false) L__M(##Miscellany, 9);
        EndActivity(PRINTING_NEWS_OF_DARKNESS_ACT);
        rfalse;
    }
    if ((previous_light_condition == true) && (lightflag == true)) {
        location = real_location;
        rfalse;
    }
    rfalse;
];
```

§**5. Silent Light Consideration.** The Adjust Light Rule makes a fuss when light changes: it prints messages, for instance. This rule is silent instead, and simply does the minimum necessary to maintain the light invariant. It is used in only four circumstances:

(a) To determine the initial light condition at start of play.
(b) When the player moves from one room to another via the "going" action.
(c) When the player moves via `PlayerTo`, which is used by "now the player is in ...".
(d) When the player changes from one persona to another.

Perhaps case (b) is surprising. Why not simply use the adjust light rule, as we do on an instance of "move player to ...", for instance? The answer is that the going action is just about to print details of the new location anyway, so it would be redundant to have the adjust light rule print out details as well.

```
[ SilentlyConsiderLight;
    lightflag = OffersLight(parent(player));
    if (lightflag) location = real_location; else location = thedark;
    rfalse;
];
```

§**6. Translucency.** `IsSeeThrough` is used at various places: roughly speaking, it determines whether `obj` being in scope means that the object-tree contents of `obj` are in scope.

```
[ IsSeeThrough obj;
    if ((obj has supporter)
        || (obj has transparent)
        || (obj has animate)
        || ((obj has container) && (obj has open)))
        rtrue;
rfalse;
];
```

§**7. Visibility Parent.** The idea of `VisibilityParent` is that it takes us from a given position in the object tree, `o`, to the next visible position above. Note that

(1) A container has an inside and an outside: this routine calculates from the "inside of `o`", which is why it returns `nothing` from an opaque closed container;
(2) Component parts are (for purposes of this routine) attached to the outside surface of a container, so that if `o` is part of a closed opaque container then the visibility parent of `o` is its actual parent.

```
[ VisibilityParent o;
    if (o && (o has container) && (o hasnt open) && (o hasnt transparent)) return nothing;
    if (o) o = CoreOfParentOfCoreOf(o);
    return o;
];
```

§**8. Find Visibility Levels.** The following routine sets the pair of variables `visibility_ceiling`, the highest visible point in the object tree above the player – or `thedark` if the player cannot see at all – and `visibility_levels`, the number of steps of `VisibilityParent` needed to reach this ceiling; or 0 if the player cannot see at all.

```
[ FindVisibilityLevels lc up;
    if (location == thedark) {
        visibility_ceiling = thedark;
        visibility_levels = 0;
    } else {
        visibility_ceiling = player;
        while (true) {
            up = VisibilityParent(visibility_ceiling);
            if (up == 0) break;
            visibility_ceiling = up;
            lc++;
        }
        visibility_levels = lc;
    }
];
```

§**9. Scope Ceiling.** Scope is almost the same thing as visibility, but not quite, and the following routine does not quite duplicate the calculation of `FindVisibilityLevels`. The difference arises in the first step, where we take the `parent` of `pos`, not the core of the `parent` of the core of `pos`: this makes a difference if `pos` is inside a container which is itself part of something else.

```
[ ScopeCeiling pos c;
    if (pos == player && location == thedark) return thedark;
    c = parent(pos);
    if (c == 0) return pos;
    while (VisibilityParent(c)) c = VisibilityParent(c);
    return c;
];
```

§**10. Object Is Untouchable.** The following routine imitates the I6 library one of the same name, but works instead by delegating the decision to the accessibility rulebook.

It is not easy to answer the question of whether someone other than the player, but in another room, can touch a multiply-present object (a two-sided door or a backdrop) and we err on the side of caution by saying yes in such cases. The question would only be asked in the case of a "try" action deliberately caused by the author, or by an instruction made by the player to someone not in the same room: in the first case, we will assume that the author knows what he is doing, and in the second case, the circumstances in which saying yes would be a wrong call are *highly* improbable.

```
[ ObjectIsUntouchable item silent_flag flag2 p save_sp decision;
    if ((p ~= player) && (LocationOf(p) ~= LocationOf(player)) &&
        ((item ofclass K4_door) || (item ofclass K7_backdrop))) {
        decision = false;
    } else {
        untouchable_object = item; untouchable_silence = silent_flag;
        touch_persona = p; if (p == actor) touch_persona = 0;
        save_sp = say__p; say__p = 0;
        if (ProcessRulebook(ACCESSIBILITY_RB, 0, true)) {
            if (RulebookSucceeded()) decision = false;
            else decision = true;
        } else decision = false;
        if (say__p == false) say__p = save_sp;
    }
    untouchable_silence = 0;
    return decision;
];
```

§11. Access Through Barriers Rule.

```
[ ACCESS_THROUGH_BARRIERS_R ancestor i j external p;
    p = touch_persona; if (p == 0) p = actor;
    ancestor = CommonAncestor(p, untouchable_object);
    if ((ancestor == 0) && (LocationOf(untouchable_object) == nothing)
        && ((untouchable_object ofclass K4_door or K7_backdrop) == false)) {
        if (touch_persona == 0) GL__M(##Take,8,untouchable_object);
        RulebookFails();
        rtrue;
    }
    ! First, a barrier between the player and the ancestor.
    if (CoreOf(p) ~= ancestor) {
        i = parent(CoreOf(p)); j = CoreOf(i); external = false;
        if (j ~= i) { i = j; external = true; }
        while (i~=ancestor && i) {
            if ((external == false)
                && (ProcessRulebook(REACHING_OUTSIDE_RB, i))
                && (RulebookFailed())) rtrue; ! Barrier
            i = parent(CoreOf(i)); j = CoreOf(i); external = false;
            if (j ~= i) { i = j; external = true; }
        }
    }
    ! Second, a barrier between the item and the ancestor.
    if (CoreOf(untouchable_object) ~= ancestor) {
        ! We can always get to the core of the item.
        i = CoreOf(untouchable_object);
        ! This will be on the inside of its parent, if its parent is a
        ! container, so there should be no exemption.
        i = parent(i); external = false;
        ! j = CoreOf(i); if (j ~= i) { i = j; external = true; }
        while (i~=ancestor && i) {
            if ((external == false) &&
                (ProcessRulebook(REACHING_INSIDE_RB, i)) &&
                (RulebookFailed())) rtrue; ! Barrier
            i = CoreOf(i);
            if (i == ancestor) break;
            i = parent(i); j = CoreOf(i); external = false;
            if (j ~= i) { i = j; external = true; }
        }
    }
    RulebookSucceeds(); ! No barrier
    rtrue;
];
```

§12. Can't Reach Inside Closed Containers Rule.

```
[ CANT_REACH_INSIDE_CLOSED_R;
    if (parameter_object has container && parameter_object hasnt open) {
        if (touch_persona == 0) GL__M(##Take,9,parameter_object);
        RulebookFails(); rtrue;
    }
    rfalse;
];
```

§13. Can't Reach Outside Closed Containers Rule.

```
[ CANT_REACH_OUTSIDE_CLOSED_R;
    if (parameter_object has container && parameter_object hasnt open) {
        if (touch_persona == 0) GL__M(##Take,9,parameter_object);
        RulebookFails(); rtrue;
    }
    rfalse;
];
```

§14. Can't Reach Inside Rooms Rule.

```
[ CANT_REACH_INSIDE_ROOMS_R;
    if (parameter_object && parameter_object ofclass K1_room) {
        if (touch_persona == 0) GL__M(##Take,14,parameter_object);
        RulebookFails(); rtrue;
    }
    rfalse;
];
```